Technology
Arts Sciences
TH Köln

Graphendatenbanken mit Neo4j

Umsetzung einer RDBM in eine Graphdatenbank

Praxisprojekt

ausgearbeitet von

Niklas Reinhardt

im Studiengang
Medieninformatik

Erster Prüfer/in: Prof. Dr. Birgit Bertelsmeier
Technische Hochschule Köln

Gummersbach, im Januar 2019

Inhaltsverzeichnis

1 Einleitung

Auf den folgenden Seiten wird es um Graphen-Datenbanken und die Umsetzung eines bestimmten Typen von Graphdatenbanken gehen. Für die Umsetzung wird eine Rationale Datenbank genommen und diese in eine Graphdatenbank überführt. Es wird auf die Unterschiedlichen Typen von Graphdatenbanken eingegangen, deren Verwendungszweck und wie diese Typen Unterschieden werden. Die Umsetzung erfolgt mit Neo4j auf dessen Funktionen und Sprache ich eingehen werde. Danach wird auf die Modellierungsart der Graphdatenbank eingegangen und wie diese von mir Realisiert wurde. Am Ende erfolgt ein Fazit und ein Ausblick wie man die von mir umgesetzte Datenbank noch verbessern und klarer gestalten kann.

Meine Informationen der verschiedenen Graphtypen erfolgt mit Hilfe des Buches „NoSQL - Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken“ von den Autoren Stefan Edlich, Achim Friedland, Jens Hampe und Benjamin Brauer (Edlich, 2010). Die Informationen über Neo4j und der Programmiersprache Cypher beziehe ich aus der Dokumentation der Offiziellen Webseite(Neo4jInc, b) und aus dem Buch „Graph Databases - New opportunities for connected date“ von Ian Robinson, Jim Webber und Emil Eifrem (Robinson, 2015).

2 Was sind Graphdatenbanken

Graphdatenbanken sind eine Sammlung von Kanten und Knoten, wo Entitäten in Knoten und deren Verbindungen mit Kanten dargestellt werden. Diese sind Teil der NoSQL-Datenbanken, welche großen Einsatz bei Big-Data-Anwendungen finden. Durch den Anstieg an der Benutzung von Smartphones, Clustern oder Clouds ist das Interesse an verteilten Anwendungen gestiegen und die damit stark ansteigende Nachfrage von vernetzten Informationen. Diese vernetzten Informationen können durch eine Graphdatenbank effizient genutzt werden. (Edlich, 2010, S.169-170)
Mit der Hilfe von Graphen ist es möglich, sonst komplizierte Datenaufrufe, die aus mehreren JOINs bestehen, mit einfacheren Graph-Traversals zu ersetzen. Es ist möglich bestimmte Graph-Algorithmen aus der Mathematik verwenden zu können, wie z.B. die Tiefensuche, um Abfragen schneller gestalten zu können. Spezielle Anwendungsfälle werden dabei gut durch ein einfaches Datenmodell unterstützt welches Whiteboard-Friendly ist.
Whiteboard-Friendly heißt, dass in einem Entwicklungsprozess die Entwickler Beispieldaten und wie diese miteinander verknüpft sind, auf ein Whiteboard zeichnen können und diese auf dem Whiteboard entstandene Modell in ein Graph-Modell umgewandelt werden kann. Es gibt verschieden Graph-Datenmodelle wie das Einfache Graph-Datenmodell, das Property-Graph-Datenmodell und den Hypergraph. Diese verschiedenen Graph-Datenmodelle stelle ich in den nächsten Kapiteln ausführlicher vor.

2.1 Einfache Graph-Modelle

Das Einfache Graph-Datenmodell basiert auf der Mathematischen Grundlage des Mathematikers Leonhard Euler, der im 18. Jahrhundert damit das Königsberger Brückenproblem mit einem Graphen formulierte. Dieses Modell ist der Grundstein für anderen Graph-Datenmodelle, die noch vorgestellt werden.
Sie bestehen, wie in Kapitel 2 erwähnt, durch Kanten und Knoten dargestellt, Kanten setzten dabei Knoten in Beziehung, die Repräsentationen Problemen aus der realen Welt sind. In einem Graphen können Beziehungen gerichtet oder ungerichtet sein. Eine gerichtete Kante stellt dabei eine einseitige Beziehung zwischen einem Knoten zu einem anderen da (siehe Abbildung 2.1), eine ungerichtete Kante stellt eine beidseitige Beziehung da. (siehe Abbildung 2.2)

Sollten Knoten durch mehr als einer Kante mit einem anderen Knoten verbunden ist dies ein Multigraph. Man kann Knoten mit Informationen anreichern, dazu kann man auch Kanten ein Gewicht zuordnen um damit einen „Gewichteten Graphen“ erstellen. (siehe Abbildung 2.3)

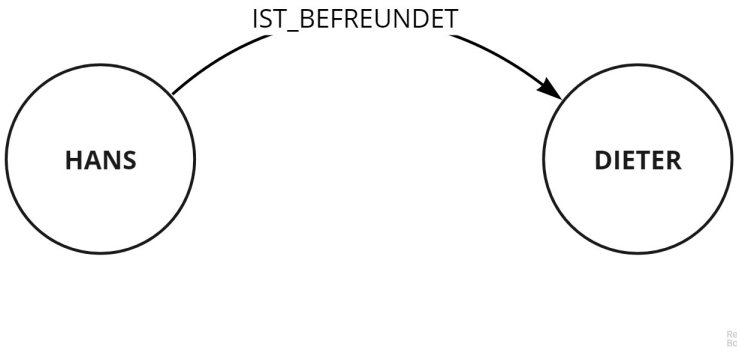

Abbildung 2.1: Gerichteter Graph

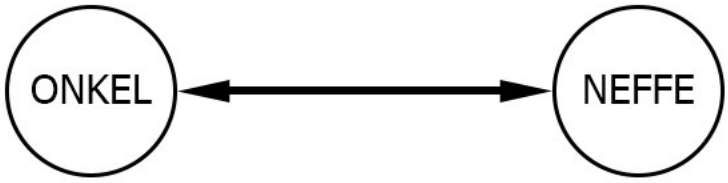

Abbildung 2.2: Ungerichteter Graph

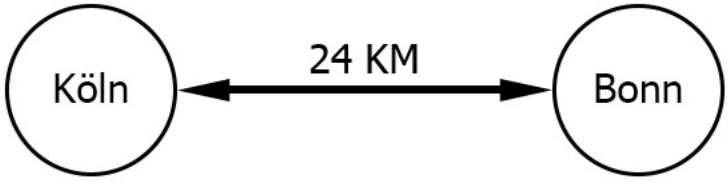

Abbildung 2.3: Gewichteter Graph

2.2 Property-Graph-Modell

Das Property-Graph-Datenmodell ist eine Erweiterung des Einfachen Graph-Datenmodell da dieses bei Problemstellungen schnell an seine Grenzen stößt. Es wird bei den meisten Graphdatenbanken auf dem Markt eingesetzt, es ist ein gerichteter, multi-relationaler Graph. Dabei bestehen die Knoten und Kanten des Graphen aus Objekten, so dass auch Kanten Properties besitzen können. Diese Properties sind Key/Value-Beziehungen wie sie in anderen NoSQL-Datenbanken eingesetzt werden. Der Key und der Value-Bereich werden vom Kanten- bzw. Knotenschema vorgegeben. Dazu bekommt jeder Knoten eine ID, um diesen eine eindeutige Identität zu geben, fast wie in einem Relationalem Datenbank Modell.
Kanten besitzen dabei einen Start- und Endknoten. Mehrfachknoten sind nur zulässig, wenn sich der Typ der Kanten unterscheidet., auch können Kanten ein Schema besitzen, welcher Kantentyp zwischen Knotentypen erlaubt ist. (Edlich, 2010, S.173)
Ein Beispiel für einen Property-Graphen sieht man in der Abbildung 2.4, dort Existieren die Knoten „Paul“, „Tom“ und „SV Rosellen“ die durch die Kanten „spielt_fuer“ und „befreundet_mit“ in Beziehung gesetzt werden.

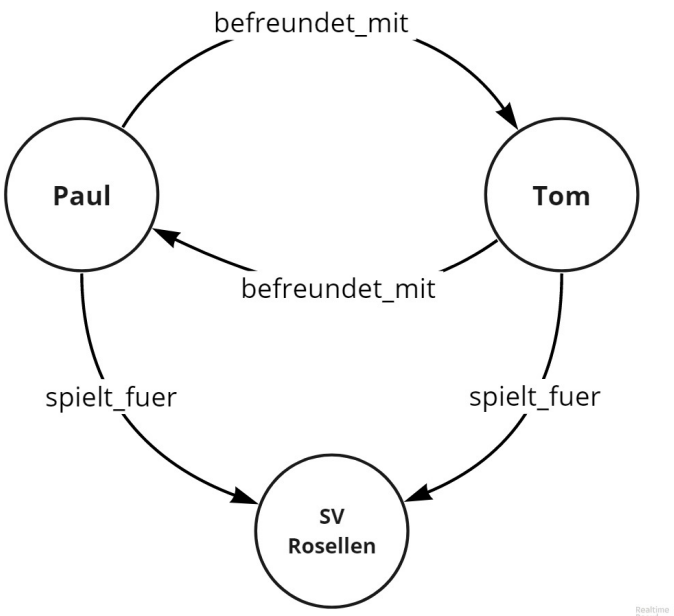

Abbildung 2.4: Beispiel für einen Property Graph

Wie man erkennt, existieren die Knoten „Paul“ und „Dieter“ die über die Relation „befreundet_mit“ Verbunden sind. Man könnte zu diesem Graphen nun die Propertis hinzufügen, dann wäre das Graph Modell ähnlich dem „Microsoft Northwind data model“

Property-Graphdatenbanken sind nahe an der Vererbungshierarchie von objektorientierten Programmiersprachen, da Kanten- und Knotentypen durch ein hierarchischen Typsystem organisiert werden. Damit wird es Entwicklern noch leichter gemacht mit Graphdatenbanken zu arbeiten.

2.3 Hypergraph Modell

Das Hypergraph-Datenmodell erweitert das Property-Graph-Datenmodell insoweit, das Kanten nun Hyperkanten sind. Diese Hyperkanten können, anders als die Kanten im Property-Graph-Datenmodell, jede Anzahl an gegebenen Knoten verbinden können. Hier gibt es keinen Start- oder Endknoten, sondern erlaubt eine Anzahl von mehreren Knoten als Ende einer Verbindung.

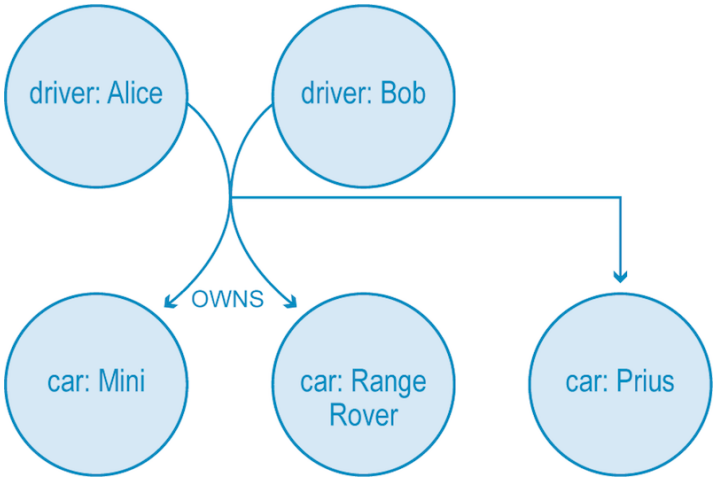

Abbildung 2.5: Beispiel für einen Property Graph. Bryce (b)

Wie die Abbildung 2.5 zeigt, müssen wir die Verbindung zwischen einem Fahrer und einem Auto nicht mehr einzeln, wie bei einem Property-Graphen, aufzeichnen, sondern können diese Verbindung direkt mit mehreren Endknoten verbinden.

Anstatt sechs Beziehungen, wie in Abbildung 2.6, zwischen den Knoten besitzen wir nur noch eine. Einen Hypergraphen kann man leicht in einen Property-Graph-Datenmodell umwandeln, in die andere Richtung ist diese jedoch nicht so einfach. Auch kann man bestimmte Informationen, die man mit den Kanten liefern könnte in einem Hypergraphen nicht darstellen, zum Beispiel ob das Auto was man besitzt das Primär Benutzte ist.
Damit ist es möglich Informationsreiche Datenmodelle zu entwickeln, welche jedoch die Gefahr bergen Details beim Modellieren zu vergessen. Innerhalb von Programmiersprachen würde man Hyperkanten als Listen oder Sets darstellen. Solche Datenmodelle finden häufig Anklang im Bereich der künstlichen Intelligenz. (Edlich, 2010, S.222)
In einem Hypergraphen können die Kanten gerichtet oder ungerichtet sein. Bryce (b)

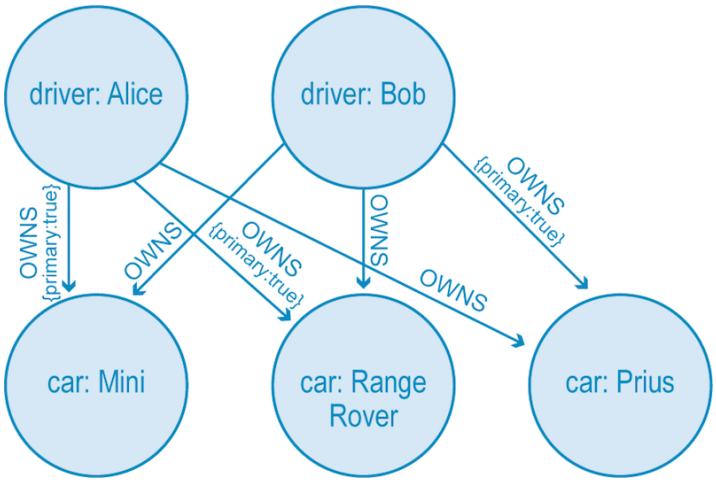

Abbildung 2.6: Beispiel für einen Property Graph. Bryce (b)

3 Was ist Neo4j?

Neo4j ist eine Open-Source-Graphendatenbank, welche nativ ist und in Java implementiert wurde. In Neo4j werden Knoten und Kanten in den internen Datenbankstrukturen als Record in den Datenbankdateien repräsentiert, weshalb sie nativ ist. Dabei baut Neo4j auf eine eigen Entwickelte Infrastruktur auf, um seine Daten zu speichern, anstatt eine andere Datenbank dafür zu nutzen.

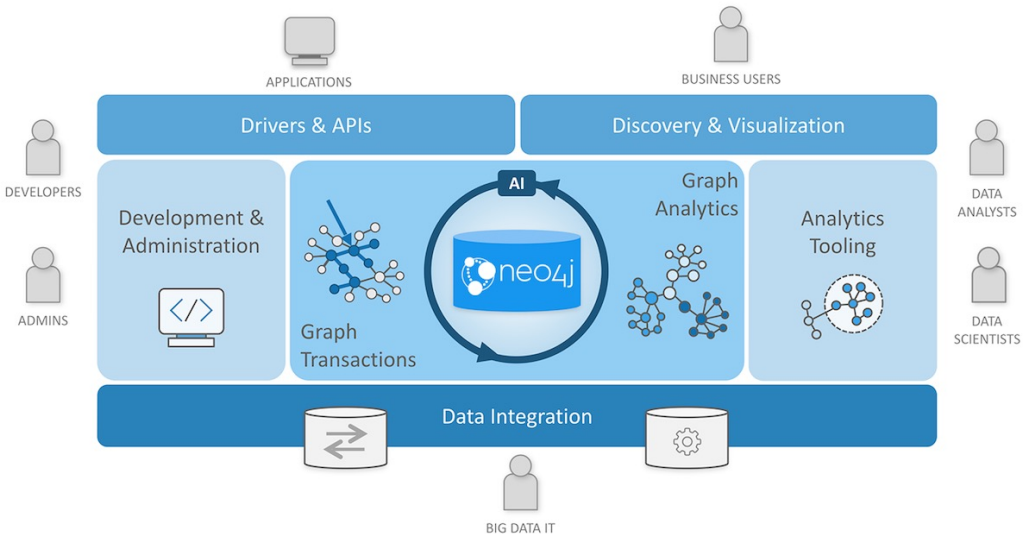

Abbildung 3.1: Komponenten der Neo4j Graph Plattform. Neo4jInc (c)

Es gibt zwei Ausführungen von Neo4j, eine Enterprise Version welche Kostenpflichtig ist und eine Community Version die frei zu erhalten ist. Die Enterprise Version setzt auf einen Server mit Hilfe des Netzwerk Protokoll BOLT und kann dadurch größere Datenmengen aufnehmen. Auch ist die Enterprise Version schneller als die Community Version, was das Importieren und arbeiten erheblich verschnellert.
Für Studenten gibt es eine Educational License, um die Enterprise Version zu benutzen, dafür muss man nur mit dem Support in Kontakt treten. Neo4j ist dabei die beliebteste Graphendatenbank auf dem Markt (Destatis) und wird von vielen bekannten Unternehmen, wie z.B. ebay oder Microsoft, eingesetzt.

Neo4j ist Schema Optional, was heißt, dass Indexes und Constraints nicht nötig sind, aber bei großen Datenmengen eingesetzt werden sollten. Neo4jInc (d) Ein großer Vorteil von Neo4j oder Graphen im Allgemeinen ist ihre große Flexibilität und Anpassbarkeit. Sollten sich die Anforderungen im Projekt ändern, kann der Graph jederzeit an die Gegebenen Umstände angepasst werden. Sollte man zum Beispiel während des

Projektes bemerken, dass der Graph den Anforderungen nicht im vollen Umfang gerecht wird, kann dieser angepasst werden und die Änderungen schnell implementiert werden.
Dies ist für Projekte mit Scrum ein großer Vorteil, da sich dort die Anforderungen mit jedem Sprint ändern können.

3.1 Welche Art von Graph-Modell wird verwendet

In Neo4j wird eine Unterart von einem Property-Graph-Modell eingesetzt, das sogenannte Labeled-Property-Graph-Modell. (siehe Abbildung 3.2) Dabei werden die einzelnen Knoten mit einem Label zu einem Set zusammengefasst. Damit ist es möglich sich alle Knoten eines Labels anzeigen zu lassen, was Abfragen erleichtert und so näher an SQL-Statements ist. So kann man sich alle Knoten mit dem Label „PERSON“ ausgeben lassen die den Namen „Tom“ haben. Dieses Label kann mit weiteren Labeln erweitert werden, um einen Knoten weiter zu spezifizieren.
Labels können auch dafür eingesetzt werden, um einen Temporären Status für Knoten festzulegen wie z.B. „:Gesperrt“ bei Twitter-Accounts, die von Twitter gesperrt wurden. Diese können dann nur durch das Löschen des temporären Labels wieder freigegeben werden.
In Neo4j muss ein Knoten jedoch kein Label haben und ein Knoten kann auch mehrere Labels haben. Beziehungen zu Duplizieren sollte in diesem Graph-Modell nur gemacht werden, wenn die Beziehung in die andere Richtung für einen bestimmten Use-Case gebraucht wird. (Neo4jInc, b)

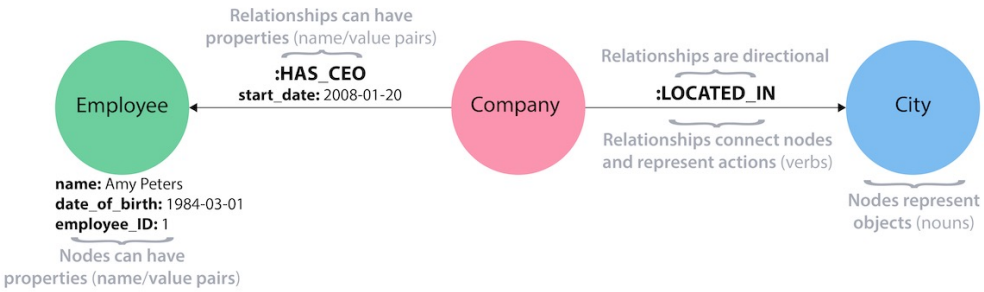

Abbildung 3.2: Beispiel für einen Labled Property Graph Neo4jInc (a)

3.2 Die Programmiersprache Cypher

Cypher ist eine Deklarative Graphen Abfragesprache, welche schnell und effizient Informationen aus einem Graphen Abfragen und Updaten kann. Dabei sind viele Abfragen von SQL inspiriert, wie z.B. „WHERE“ und „ORDER BY“. Sie wurde für Neo4j entwickelt, hat sich dann aber zu einem OpenSource Projekt entwickelte, welches von mehreren Kommerziellen Datenbank Produkten benutzt wird.
Cypher setzt dabei auf die englische Sprache und einer einfachen Symbolik, die es Entwicklern erlaubt Abfragen einfach zu schreiben und zu lesen. Das Datentypsystem in Cypher kann Strings, Integer, Listen, Sets, Maps, Gleitkommazahlen, Booleans und Arrays aufnehmen. Auch Datum und Zeit können in Cypher gespeichert werden und werden häufig bei Beziehungen benutzt. (Neo4jInc, b) Hauptbestandteile einer Abfrage in Cypher sind „MATCH“, „WHERE“ und „RETURN“.

```
MATCH p=(n:PERSON)-[:KNOWS]-() WHERE n.name = "Tom" RETURN p;
```

Abbildung 3.3: Beispiel für eine MATCH Abfrage

Die Abfrage in der Abbildung 3.3 zeigt, wie man in Cypher die Nachbarknoten mit der Beziehung „:KNOWS“, für den Knoten mit dem Namen ”´Tom“ anzeigen lassen kann.
Will man einen Graphen Updaten, kann dies durch die Befehle „CREATE“, „DELETE“, „SET“, „REMOVE“ oder „MERGE“ passieren. Der Befehl „MERGE“ schaut dabei erst ob ein Knoten schon existiert, sollte dies nicht der Fall sein wird der Knoten erstellt, und kann so dann zum Beispiel eine neue Beziehung erstellen. Eingesetzt werden sollte „MERGE“ jedoch am besten mit „UNIQUE CONSTRAINTS“, damit sichergestellt wird, dass ein Knoten wirklich nur einmal in der Datenbank existiert.
Mit „CREATE“ erstellt man eine neue Kante oder einen neuen Knoten, mit „DELETE“ löscht man einen Knoten oder eine Kante, mit „REMOVE“ löscht man ein Label oder eine Property und SET setzt eine Property oder ein Label.

In Cypher ist es möglich, CSV Dateien, die man vorher aus anderen Datenbanken exportiert hat, zu importieren, welches über den Befehl „LOAD CSV“ funktioniert. Dabei kann die Datei auf einem Webserver liegen, oder auch im Filesystem selbst. Beim Import kann man nun Befehle wie „MERGE“, „MATCH“ oder „CREATE“ ausführen. Dieser Import ist sehr schnell, hier sollte man jedoch trotzdem vorher einen Index oder ein Constraint einsetzten, um diesen so effizient und schnell wie möglich durchzuführen. Auch „SET“ kann ausgeführt werden, um nur einzelne Werte aus einer CSV Datei zu bestehenden Knoten oder auch Kanten hinzuzufügen.

In diesem Beispiel (Abbildung 3.4) sehen wir, wie der einfache Import mit „CREATE“ in Cypher funktioniert. Wie man sieht ist es relativ einfach zu erkennen was passiert, man erstellt einen Knoten mit dem Label „:PERSON“ und übergibt diesem die Werte „name“ und „alter“. Man könnte hier auch direkt eine Kante erstellen um eine Person mit einer bestehenden Person zu Verknüpfen. Sollte man eine zu große

```
LOAD CSV WITH HEADERS FROM "file://example.csv" AS row
CREATE(t:EXAMPLE { name: row.name, year: row.date});
```

Abbildung 3.4: Beispiel für einen CSV Import

CSV Datei Importieren wollen, sollte der Befehl „USING PERIODIC OMMIT“ gesetzt werden, der immer nur 1000 Reihen Commitet, das reduziert den Speicheraufwand der Transaktion.
Der Import ist sehr mächtig und die wichtigste Methode, um andere Datenbank-Modelle in einen Graphen zu überführen. Es gibt auch so genannte ETL Tools, die die Umwandlung von RDBMS-Systemen in einen Graphen zu erleichtern, was aber dazu führen kann das nicht jeder Use-Case abgedeckt wird.

3.3 SQL-Befehle in Cypher

Sieht man sich Cypher-Statements an, kann man erkennen das eine nähe zu SQL nicht zu leugnen ist. Anstatt einem „SELECT“ und „FROM“ Statement in SQL benutzt man in Cypher den ” ´MATCH“-Befehl von Cyper. In diesem Statement kann angeben welchen Knoten mit welcher Beziehung man anzeigen möchte. Damit verzichtet Cypher auf das ” ´JOIN“ Statement aus SQL, da man die Beziehung von zwei Knoten mit unterschiedlichem Label direkt angibt.

```
SELECT name FROM Person
LEFT JOIN Person_Department
  ON Person.Id = Person_Department.PersonId
LEFT JOIN Department
  ON Department.Id = Person_Department.DepartmentId
WHERE Department.name = "IT Department"



MATCH (p:Person)<-[:EMPLOYEE]-(d:Department)
WHERE d.name = "IT Department"
RETURN p.name
```

Abbildung 3.5: Beispiel einer „JOIN“-Abfrage und einer „MATCH“-Abfrage

Wie man in den Abbildung 3.5 sieht, kann man mit weniger Code die gleiche Information bekommen, wie in SQL. Durch die fehlenden „JOIN“ ist die Gefahr von Fehlern reduziert und der Befehl ist in Cypher schneller geschrieben und wird schneller ausgeführt.

Das Erstellen von Tabellen wird in SQL wie in Abbildung 3.6 ausgeführt und mit Daten gefüllt.

```
CREATE TABLE recipes (
  recipe_id INT NOT NULL,
  recipe_name VARCHAR(30) NOT NULL,
  PRIMARY KEY (recipe_id),
  UNIQUE (recipe_name)
);

INSERT INTO recipes
    (recipe_id, recipe_name)
VALUES
    (1,"Tacos"),
    (2,"Tomato Soup"),
    (3,"Grilled Cheese");
```

Abbildung 3.6: Beispiel ” ´CREATE TABLE“-Befehl in SQL

```
CREATE (a:RECIPE {Name: "Tacos"}),(b:RECIPE {Name: "Tomato Soup"}), (c:RECIPE {Name: "Grilled Cheese"})
Return a, b, c
```

Abbildung 3.7: Beispiel ” ´CREATE“-Befehl in Cypher

In Abbildung 3.7 sehen wir wie die gleiche Tabelle mit den gleichen Daten in SQL funktioniert. Dabei ist der schreib Aufwand bei Cypher und in SQL effektiv nicht so viel größer. Jedoch empfinde ich die „CREATE“-Methode einfacher als die von SQL, da das Statement einfacher zu lesen und bei Fehlern anzupassen ist.

Anbei in Abbildung 3.8 noch ein paar Befehle wie sie in SQL gemacht werden und wie diese in Cypher umgesetzt wurden.

```
IN SQL:

SELECT p.*
FROM products as p;

IN CYPHER:

MATCH (p:Product)
RETURN p;

IN SQL:

SELECT p.ProductName, p.UnitPrice
FROM products as p
ORDER BY p.UnitPrice DESC
LIMIT 10;

IN CYPHER:

MATCH (p:Product)
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice DESC
LIMIT 10;

IN SQL:

SELECT p.ProductName, p.UnitPrice
FROM products AS p
WHERE p.ProductName = 'Chocolade';

IN CYPHER:

MATCH (p:Product {productName:"Chocolade"})
RETURN p.productName, p.unitPrice;

IN SQL:

SELECT p.ProductName, p.UnitPrice
FROM products as p
WHERE p.ProductName IN ('Chocolade','Chai');

IN CYPHER:

MATCH (p:Product)
WHERE p.productName IN ['Chocolade','Chai']
RETURN p.productName, p.unitPrice;
```

Abbildung 3.8: Befehle in SQL und dann in Cypher

4 Modellierung

Die Modellierung eines Graphen findet meist schön während den Vorbereitungen der Entwicklung statt, wenn in einem Projekt die Daten und wie diese in Verbindung stehen besprochen wird. Dabei besteht eine enge Affinität zwischen dem Logischen und Physikalischen Model.
Wie schon erwähnt sind Graphen sehr Whiteboard-Friendly, was die Modellierung erleichtert. Da dieses Model, welches wir in der Kreativen und Analytischen Phase aufzeichnen, sehr nahe dem kommt, was wir am Ende in unsere Datenbank implementieren. (Robinson, 2015, S.26)
Durch die einfache Verständlichkeit eines Graphen, verstehen auch nicht Entwickler welche Daten es gibt und wie diese in Verbindung stehen. Dabei ist der Graph sehr flexibel und kann jederzeit angepasst werden.
Im Gegensatz zu einem ER-Modell müssen keine JOIN-Tabellen erstellt werden, um die Daten abfragen bei Normalisierten Tabellen zu ermöglichen. Dabei kann man viele Domänen mit einem relationalem oder Objektorientierten Modell zu einem Graphen überführen.

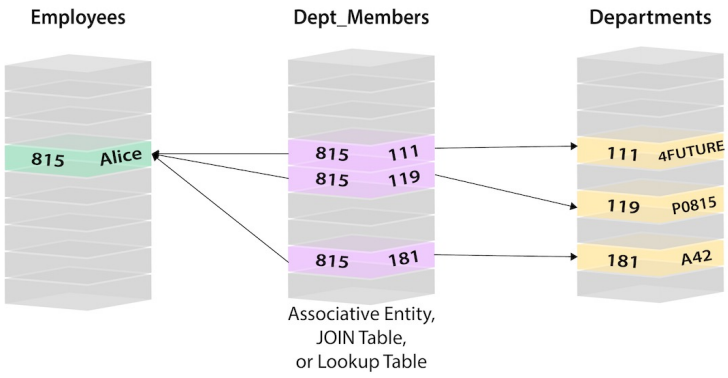

Abbildung 4.1: Das Relationale Modell einer Datenbank und deren Zuordnung der Daten.Neo4jInc (a)

Dabei geht es nicht nur Grundlegend darum wie die Entwickler denken wie Dinge in der Domäne miteinander kommunizieren, sondern auch welche konkreten Fragen wir an diese Domäne stellen.
Dazu kann vor der Modellierung eines Graphen Use-Cases erstellt werden um festzustellen welche Anforderungen man an das System hat und welche Daten dabei abgefragt werden sollen. Daraus kann man dann Fragen an die Domäne formulieren, die die

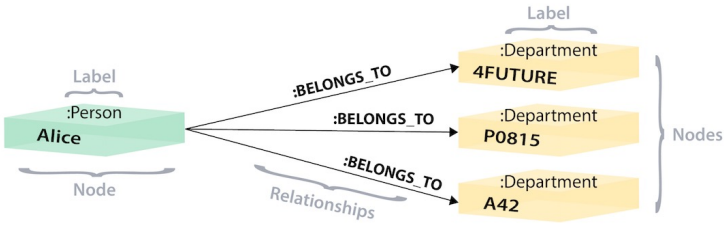

Abbildung 4.2: Das Graph Modell einer Datenbank und deren Zuordnung der Daten. Neo4jInc (a)

Erstellung eines Graphen erleichtern und es allen an der Entwicklung teilnehmenden Menschen erleichtert die Anforderungen an die Domäne zu verstehen. Diese Fragen sollen dabei auch helfen, die reichhaltigen Beziehungen zwischen den Daten darzustellen. Während der Modellierung eines Graphen geht man gerne von konkreten Daten und Anwendungsfällen aus. Dadurch kann man schnell erkennen ob die Konkreten Fälle das Model Aussagekräftig ist und die Beziehungen so modelliert sind das alle gewünschten Szenarien unterstützt werden können. (Hunger, 2014, S.45)

4.1 Unterschiede zwischen ER-Diagramm und Graph Modell

Die größten Unterschiede zwischen dem ER-Diagramm und dem Graph Modell ist die Art der Erstellung. Wie schon beschrieben, kann man ein Graph Modell direkt aus dem Datenmodell, welches man auf einem Whiteboard malt übernommen werden. Bei einem ER-Diagramm wird das Datenmodell in Entitäten, Beziehungen und Attributen unterteilt und daraus das ER-Diagramm erstellt. Dabei wird bei den Beziehungen versucht herauszufinden, welche Kardinalitäten es gibt. Dies erfordert von den Entwicklern dieses Modells nochmal einen entsprechenden Workload, um ein Modell zu entwickeln, welches alle Daten für dieses System anzeigen kann. So werden bei n:m-Beziehungen zum Beispiel eine Tabelle erstellt, welche den Primärschlüsseln beider Tabellen enthält, um wieder eine 1:n-Beziehung darzustellen, da es nicht möglich ist eine n:m-Beziehung in einer relationalen Datenbank umzusetzen. (Jarosch, 2010, S.200)
In einem Graph Modell ist es nicht nötig mit Primär- oder Fremdschlüsseln zu arbeiten, da die Graphen Logisch über deren Namen miteinander verknüpft werden können. Sollten trotzdem einmal Daten auftauchen, welche nicht durch einen Namen eindeutig benannt werden können, kann man diesen Daten eine ID geben, welche jedoch dann nicht als Fremdschlüssel in einem anderen Datensatz auftauchen müssen, um die Beziehung zwischen den beiden herzustellen.

4.1.1 Darstellung Entitäten im Graph Modell

Entitäten aus dem Relationalem Modell werden im Graph Modell als Knoten umgesetzt. Dabei wird der Entitätstyp als Label für die Knoten genommen. Hat man zum Beispiel den Entitätstyp „Schüler“ mit den Entitäten „Julia, Nicole, Axel“ werden diese zu Knoten überführt. Dies wird in der Abbildung 4.3 veranschaulicht. Dabei können Attribute der Entitäten im Graph Modell wegfallen, wie ein Fremd- oder der Primärschlüssel.

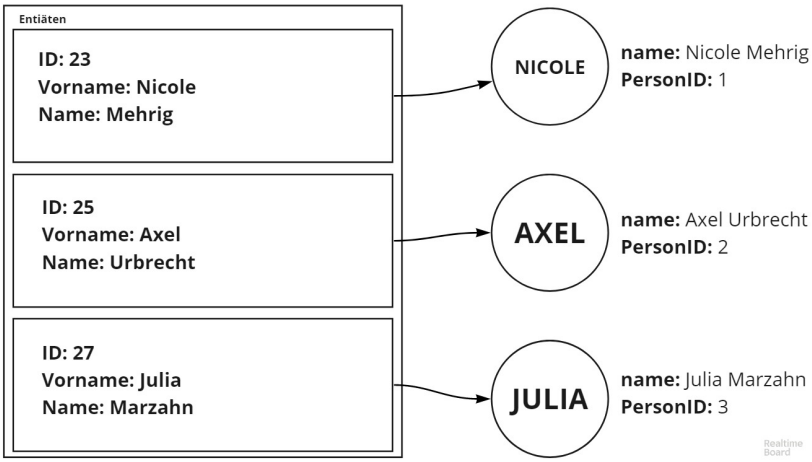

Abbildung 4.3: Entitäten die zu Knoten werden.

Man sollte Knoten immer durch ein Label zusammenfassen, welches sich aus den Entitätstypen eines ER-Diagramm ergeben, um dadurch die Modellierung zu erleichtern. Sollte man Knoten nicht durch ein Label zusammenfassen, kann es bei einer zu großen Menge an Entitäten unübersichtlich werden. Ein negatives Beispiel dazu zeigt Abbildung 4.4, ohne die Farben die die Labels zeigen, wäre eine Unterscheidung nicht möglich.

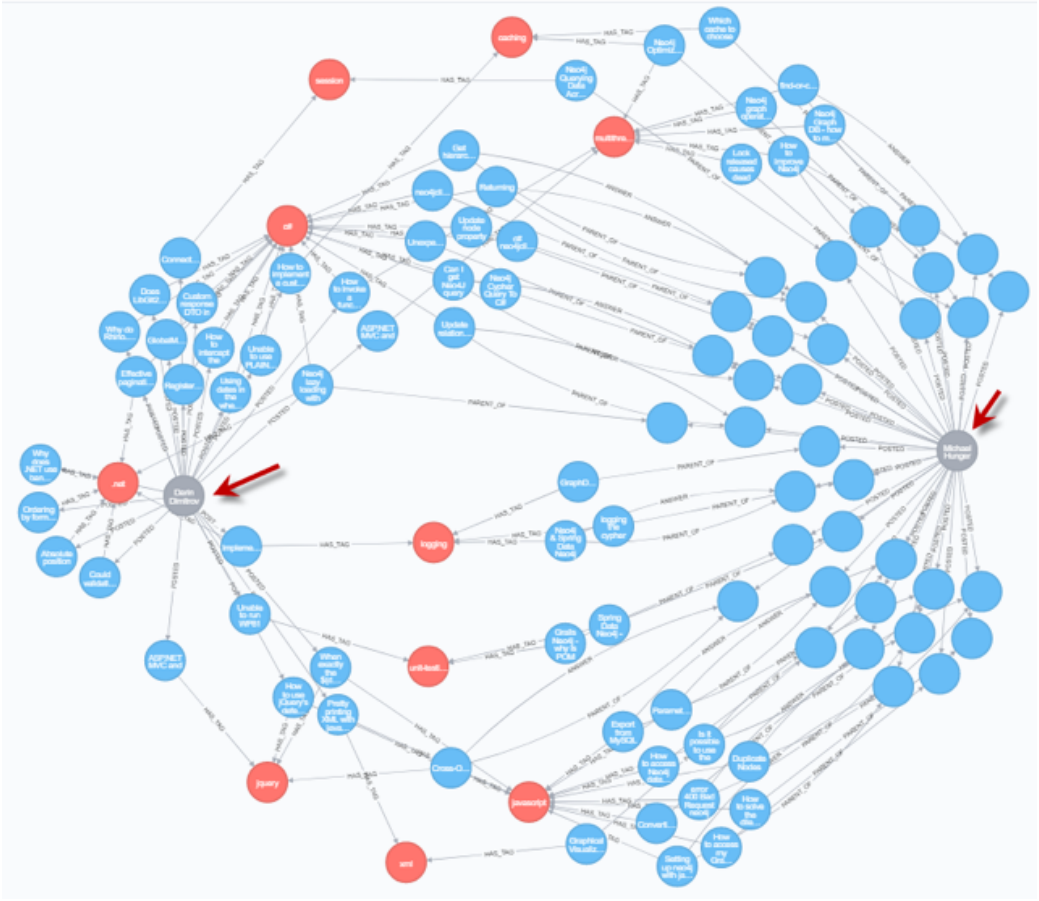

Abbildung 4.4: Beispiel einer Datenbank mit vielen Knoten. Yu

4.1.2 Darstellung von Relationen im Graph Modell

Relationen im Graph Modell wird als Kante zwischen einem Start- und Endknoten dargestellt, sie sind also immer gerichtet. Dabei kann die Kante auch Properties besitzen welche, die Kante wird mit der Art wie die Knoten in Verbindung stehen beschrieben, wie zum Beispiel mit „:KNOWS“.

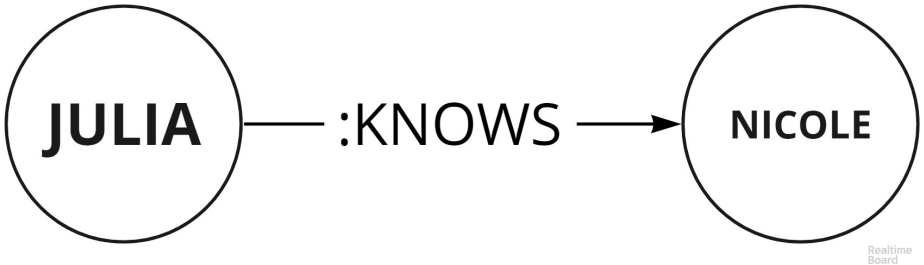

Abbildung 4.5: Beispiel Knoten mit Relation „:KNOWS“

In ER-Diagrammen ist die Form der Notation von Beziehungen wichtig, hat man sich dort für eine Notationsform, wie zum Beispiel die Chen-Notation, sollte man diese im Kompletten Diagramm beibehalten. Da eine solche Notation im Graph Modell für Kanten weg fällt, muss man sich nicht für eine Notationsform entscheiden.
Fremdschlüssel-Beziehungen im ER-Diagramm werden im Graph Modell in eine Beziehung umgewandelt, so wie Entitäten die nur durch eine n:m-Beziehung entstanden sind.

4.1.3 Graph Modellierung und Probleme mit dieser

Das Graph Modell ist passend für Umgebungen wo Daten mit vielen JOINs abgerufen werden müssen, wie zum Beispiel CM-Systeme oder Soziale Netzwerke. Es ist jedoch immer zu beachten das nicht jeder Anwendungsfall für geeignet ist, um einen Graphen dafür zu erstellen. Sind Daten zum Beispiel sehr tabellarisch und gut strukturiert, bietet sich eher eine Oracle Datenbank an. Aber auch eine Kombination aus Oracle und Neo4j wäre für viele Problemdomänen eine gute Wahl. Dabei kann man Neo4j für Suchvorgänge benutzen und Oracle dafür, um die Transaktionsdaten auszulesen.
Beispiele für Datenbanken mit strukturierten und tabellarischen Daten wären zum Beispiel Statistiken. Würde man zum Beispiel versuchen die Daten von der Statistikseite destatis (Destatis) in einem Graph Modell umzusetzen, wäre dies ein sehr schwieriges Unterfangen, da Statistiken oft sehr strukturierte und Tabellarische Daten sind. Als Beispieltabelle könnte man mit dem folgenden Aufruf aus Abbildung 4.9 nehmen.

Eine Umsetzung dieser Daten als Graph Modell wäre wohl nicht im vollen Umfang möglich und eine Relationale Lösung mit einem ER-Diagramm würde sich mehr für eine Lösung anbieten.

Auch durch die Einfachheit eines Graphen kann es passieren, dass man wichtige Zusammenhänge vernachlässigt oder nicht bemerkt, die durch eine genauere Analyse wie sie bei der Erstellung eines ER-Diagramms vorgenommen wird, nicht passiert würden. Natürlich lässt sich der Graph jederzeit anpassen, aber dies erfordert eine erneute Iteration des Graph Modells. Diese Iteration passiert bei relationalen Datenbanken meistens schon bevor diese eingesetzt werden.

Ein Beispiel liefert dafür Bryce Merkl Sasaki, ein Graphdatenbank Spezialist, der während seiner Arbeit für einen Kunden eine Fraud Detection für Email Kommunikation entwickelte und dabei mit seinem Team ein einfaches Graph Modell entwickelte bei dem sich später rausstellte, dass der Emailtext nicht übergeben wurde. (siehe Abbildung 4.6)

Dies geschah dadurch, dass man im Englischen den Satz „Bob sent an email to Charlie“ in die Phrase „Bob emailed Charlie“ umgewandelt hat. Durch diese Abänderung machte man den Modellierungsfehler, aus Abbildung Abbildung 4.6. Bryce und sein Team entwickelten daraufhin ein stärkeres Modell, welches dann auch die Email und deren Inhalt zurückgeben konnte. (siehe Abbildung 4.7)
Die E-Mail wurde dabei als neuer Knoten mit eigenen Beziehungen implementiert. Aber auch dieses Modell war nicht optimal, da man nicht die Antworten auf Emails

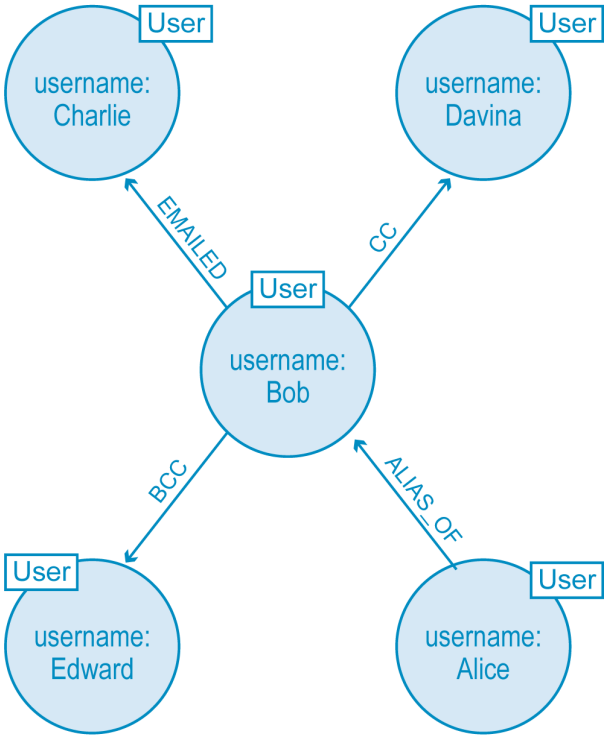

Abbildung 4.6: Fehlerhafte Gestaltung der Fraud Detection. Bryce (a)

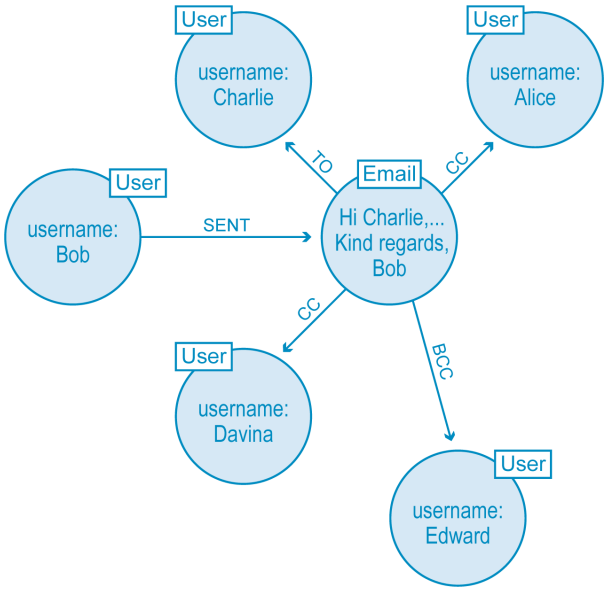

Abbildung 4.7: Überarbeitung der Fraud Detection. Bryce (a)

im Modell erkennen konnte. Also wurde das überarbeitete Modell noch mal von Bryce und seinem Team angepasst.
Man fügte einfach neue Kanten Bezeichnungen ein, mit „FORWARDED“ und „RE-

PLIED_TO“. Wie man den dem Beispiel sieht, kann die falsche Formulierung von Anforderungen, Use Cases oder auch Aussagen in der Domäne zu einer fehlerhaften Modellierung führen.

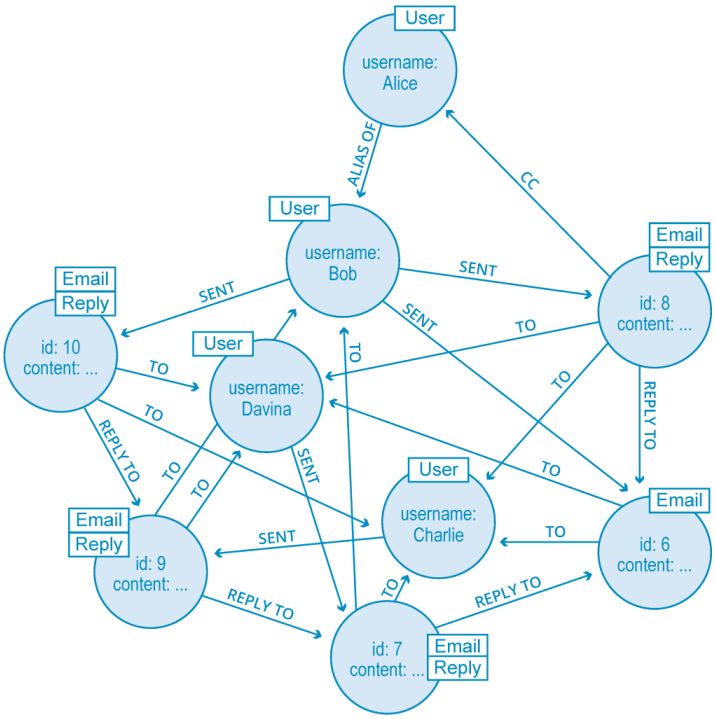

Abbildung 4.8: Finale Version der Fraud Detection. Bryce (a)

Bevölkerung, Erwerbstätige, Erwerbslose, Erwerbspersonen, Nichterwerbspersonen [jeweils im Alter von 15 bis unter 65 Jahren]: Deutschland, Jahre, Geschlecht

Mikrozensus
Deutschland

Stichmonat / Jahr	Bevölkerung	Erwerbstätige	Erwerbslose	Erwerbspersonen	Nichterwerbspersonen
	1000	1000	1000	1000	1000
männlich					
05/2000	28 067	20 439	1 997	22 433	5 634
04/2001	28 014	20 375	2 051	22 426	5 588
04/2002	27 927	20 073	2 285	22 358	5 570
05/2003	27 824	19 720	2 626	22 345	5 479
03/2004	27 687	19 397	2 842	22 239	5 448
2011	26 415	20 429	1 333	21 762	4 653
2012	26 493	20 559	1 233	21 792	4 702
2013	26 560	20 632	1 227	21 859	4 701
weiblich					
05/2000	27 367	15 793	1 726	17 517	9 850
04/2001	27 297	16 040	1 678	17 718	9 579
04/2002	27 303	16 045	1 779	17 824	9 479
05/2003	27 234	16 014	1 988	18 002	9 232
03/2004	27 084	15 812	2 100	17 911	9 174
2011	26 204	17 741	1 061	18 802	7 402
2012	26 246	17 839	985	18 823	7 423
2013	26 288	18 090	947	19 037	7 251
Insgesamt					
05/2000	55 433	36 232	3 722	39 950	15 483
04/2001	55 312	36 415	3 730	40 144	15 168
04/2002	55 230	36 118	4 064	40 182	15 049
05/2003	55 058	35 734	4 614	40 347	14 711
03/2004	54 771	35 209	4 941	40 150	14 621
2011	52 619	38 170	2 394	40 564	12 055
2012	52 739	38 398	2 218	40 615	12 124
2013	52 848	38 722	2 173	40 896	11 952

Bevölkerung, Erwerbstätige, Erwerbslose, Erwerbspersonen,
Nichterwerbspersonen: im Alter von 15 bis unter 65 Jahren
(bis 1971: Geburtsjahrmethode, ab 1972: Altersjahrmethode).

Bis 1962: Ohne Soldaten.

1983, 1984: EG-Arbeitskräftestichprobe.

1987: Revidierte Hochrechnung (Basis Volkszählung 1987).

Bis 1990: Früheres Bundesgebiet.

Ab 2005:
Umstellung des Mikrozensus von einer Erhebung mit fester
Berichtswoche auf eine kontinuierliche Erhebung mit
gleitender Berichtswoche.

Ab 2011:
Hochrechnung anhand der Bevölkerungsfortschreibung auf Basis
des Zensus 2011.

Ab 2016:
Aktualisierte Auswahlgrundlage der Stichprobe auf Basis des
Zensus 2011.

Ab 2017:
Ohne Bevölkerung in Gemeinschaftsunterkünften.

Abbildung 4.9: Aufruf des Statistikamt. Destatis

4.2 Modellanalyse des Beispiel ER-Diagramm

Von der Technischen Hochschule habe ich die Datenbank der Webseite IMDB, welche sich auf Filme und Serien spezialisiert, bekommen. Diese wurde von mir in einen Graphen überführt, dafür war zu erst die Erstellung eines Graphen Modells nötig.
Dafür ist eine Analyse des vorhandenen Relationalen Modell (siehe Abbildung im Anhang) nötig, welches man mit dem Data Modeler von Oracle erstellen konnte. Aus diesem Graph Modell habe ich ein Domänenmodell erstellt, welches man in Abbildung 4.10 sieht.

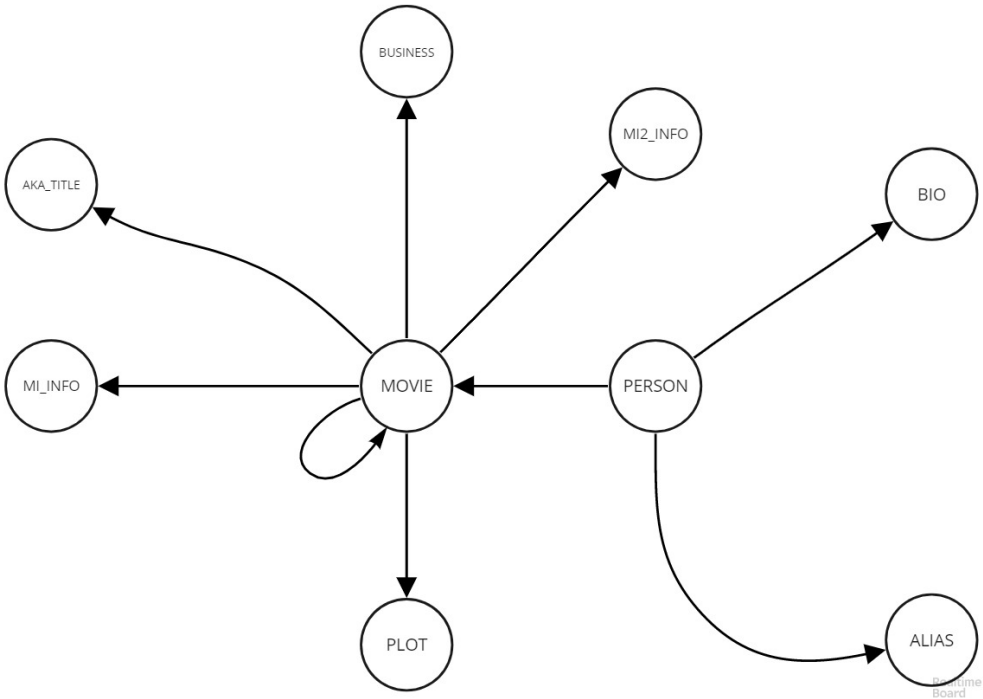

Abbildung 4.10: Domänenmodell zum Beispiel ER-Diagramm

4.2.1 Analyse der Entitäten

Zuerst habe ich versucht die Entitäten aus dem Beispiel ER-Diagramm zu ermitteln, welche für eine Umwandlung in einen Knoten für geeignet hielt. Die Hauptentitäten dafür sind schnell ersichtlich und zwar die Entitäten „IMDB_MOVIE“ und „IMDB_PERSON“.
Weiterhin wurden Tabellen genommen die Informationen über die Entität „IMDB_MOVIE“ enthalten, bei denen mehrere Einträge mit der gleichen „MOVIE_ID“ vorhanden waren, wie „IMDB_MOVIE_INFO“ oder „IMDB_MOVIE_INFO_2“. Auch „IMDB_BUSINESS“ wurde als Entität von mir Analysiert die zu einem Knoten gemacht werden konnten. „IMDB_PLOT“ wurde heraus analysiert, da einige Attribute mehrfach zu einem Film gehören konnten und ich kein Array von großen Zeichenketten haben wollte. Da ein

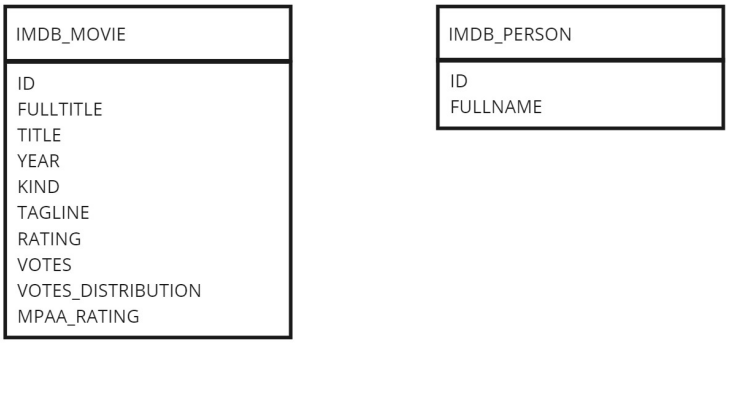

Abbildung 4.11: Die Entitäten „IMDB_MOVIE“ und „IMDB_PERSON“

Film mehrere Titel haben konnte und ich erkenntlich machen wollte in welcher Sprache diese sind, wurde auch die Entität „IMDB_AKA_TITLE“ heraus analysiert. In der Abbildung 4.12 sieht man welche Entitäten, die eng mit „IMDB_MOVIE“ zusammengehören, von mir heraus analysiert wurden.

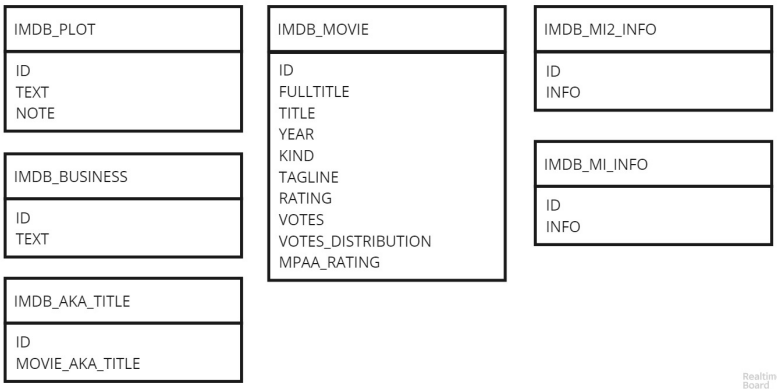

Abbildung 4.12: Die Entitäten die mit „IMDB_MOVIE“ in Verbindung stehen sollen.

Als restliche Entitäte habe ich „IMDB_AKA_NAMES“ und „IMDB_BIO“ gefunden, diese hängen eng mit „IMDB_PERSON“ zusammen. In der Abbildung 4.14 zeige ich alle Entitäten die eng zu „IMDB_PERSON“ gehören.

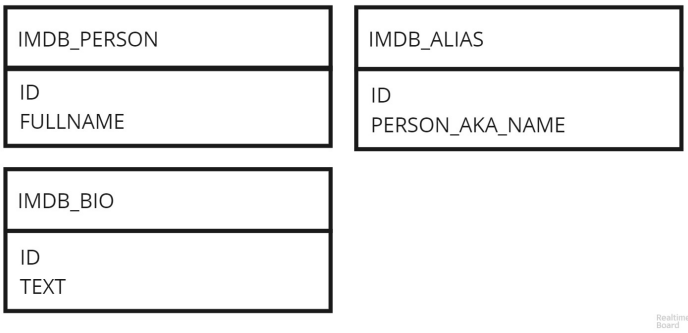

Abbildung 4.13: Die Entitäten die mit „IMDB_PERSON“ in Verbindung stehen sollen.

4.2.2 Analyse der Relationen

Die Relationen aus dem ER-Diagramm zu analysieren war um einiges schwieriger. Als eine der Hauptrelationen wurde von mir die Relation zwischen „IMDB_PERSON“ und „IMDB_MOVIE“ gesehen. Da eine Relation zwischen beiden nur durch die Entität „IMDB_CAST“ statt, weshalb diese für mich als eine „Relations-Entität“ festgelegt wurde. Zu dieser wurde die Entität „IMDB_SLV_ROLETYPE“ hinzugefügt, um die Relation mit allen Daten auszustatten.

Meistens habe ich mich bei den Relationen an die Primär- und Fremdschlüssel Relationen gehalten und diese als Relationen gesehen die ich später als wichtig erachte. Am Ende habe ich folgende Relationen aus den Tabellen in der Abbildung 4.14 heraus analysiert.

Als Relationen, die auf dieselbe Entität zeigen und vorhanden sein müssen habe ich „IMDB_SLV_KIND_MOVIE“ und „IMDB_SLV_MOVIELINK“ heraus analysiert. Wie man auch in der Abbildung sieht, zeigen beide auf „IMDB_MOVIE“. Ich habe mein Modell in ein Domänenmodell überführt, um daraus später einfacher einen Graphen zu erstellen. Das Domänenmodell ist in der Abbildung 4.10 zu sehen.

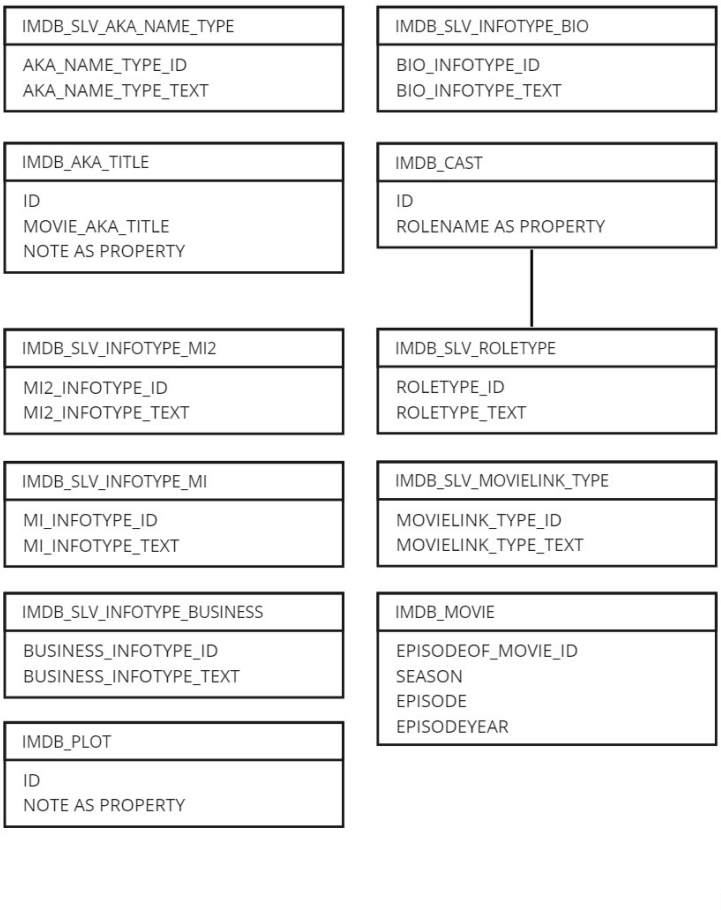

Abbildung 4.14: Die Relationen aus der IMDB Datenbank.

4.3 Modellierung des Graphen

Nachdem ich alle Entitäten und Relationen analysiert hatte, habe ich ein Graphen Modell auf dieser Grundlage erstellt. Wie man sehen kann habe ich den Graphen zuerst mit den Labeln zur besseren Übersicht erstellt. Den fertigen Graphen kann man in Abbildung 4.15 sehen und warum ich welche Kante und welchen Knoten genommen habe, wird in den anderen Kapiteln erklärt.

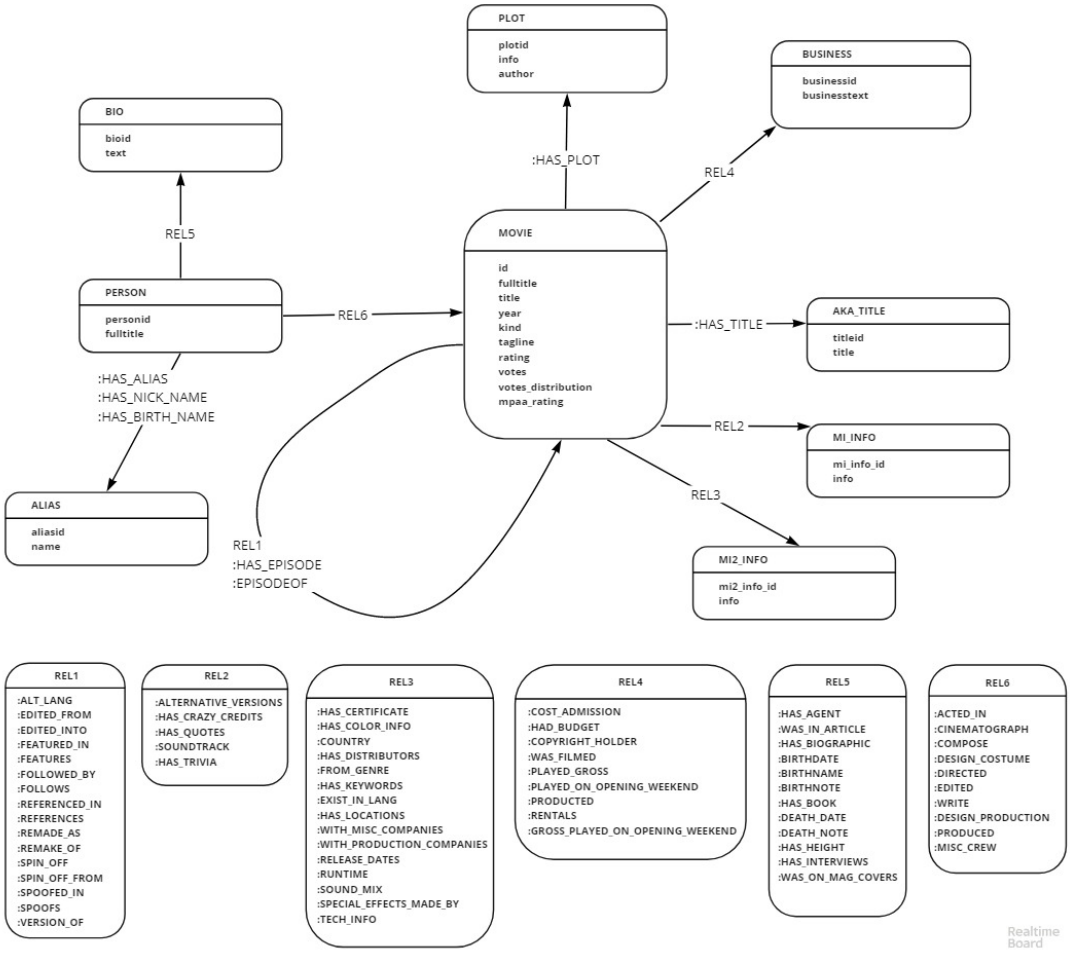

Abbildung 4.15: Der fertige Graph

4.3.1 Übertragung Entitäten in Knoten

Wie in Unterabschnitt 4.2.1 habe ich alle Entitäten in Knoten übertragen. Heraus kamen die Knoten mit den Labels „PERSON“, „MOVIE“, „BIO“, „ALIAS“, „AKA_TITLE“, „MI_INFO“ und „MI2_INFO“.

Nachdem ich die Entitäten aus dem ER-Diagramm analysiert habe und ich die Knoten festgelegt habe, legte ich die Properties fest die für die Knoten existieren würden. Diese habe ich in Tabellen festgehalten, einige der Tabellen folgen in Abbildung 4.17 und Abbildung 4.18, der Rest ist als Anhang angehängt.

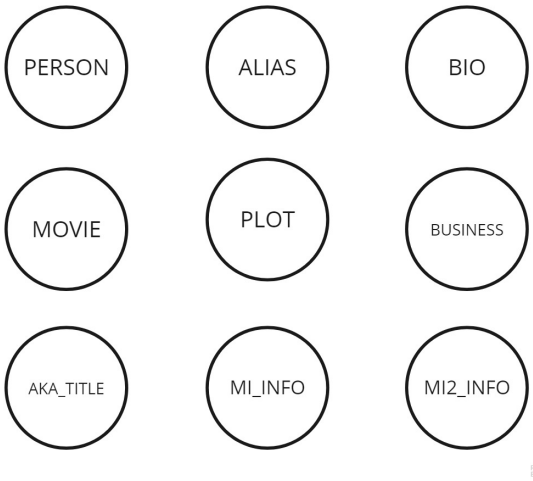

Abbildung 4.16: Alle Knoten des Graphen.

Propertyname	Property Beschreibung	Datentyp
personid	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, weil der der Name der Person mit Leerzeichen bei mir zu Fehlern geführt hat beim Erstellen der Kanten.	Integer
name	Enthält den Namen der Person.	String

Abbildung 4.17: Tabelle PERSONEN Knoten

Propertyname	Property Beschreibung	Datentyp
id	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, weil der Titel nicht eindeutig genug wäre und ich das Problem mit den Sonderzeichen nicht lösen konnte.	Integer
fulltitle	Der volle Titel des Films.	String
title	Dieser Titel enthält einen kurzen Titel.	String
year	Enthält das Jahr.	String
kind	Enthält die Art des Filmes.	String
tagline	Ist die Tagline des Filmes.	String
rating	Gibt das momentane Rating des Filmes an.	String
votes	Die Votes an die abgegeben wurden.	String
votes_distribution	Gibt die Verteilung der Votes an.	String
mpaa_rating	Ist die Alterseinstufung des Filmes.	String

Abbildung 4.18: Tabelle MOVIE Knoten

4.3.2 Übertragung Relationen in Kanten

Nachdem die Entitäten in Knoten übertrugen wurden, mussten noch die Kanten erstellt werden, die die Knoten in Beziehung setzen. Um die Knoten „PERSON“ und „ALIAS“ in Beziehung zu setzen, habe ich die Beziehungen „:HAS_ALIAS“, „:HAS_BIRTHNAME“ und „:HAS_NICK_NAME“ erstellt, wie in Abbildung 4.19 zu sehen. So konnte man zwischen den Knoten sehen welche Art von Alias diese haben.

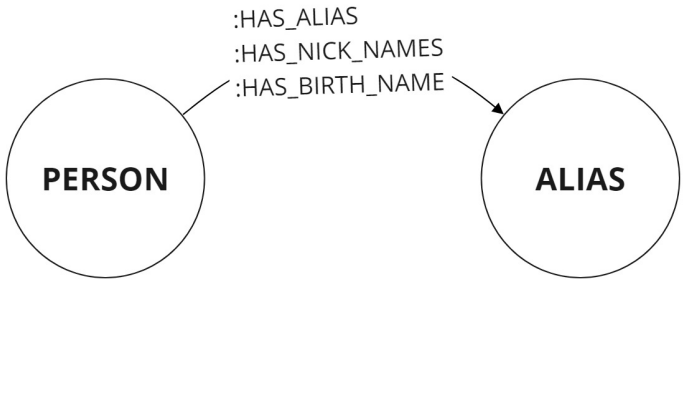

Abbildung 4.19: Kante zwischen „ALIAS“ und „PERSON“

Eine wichtige Beziehung ist die Zwischen „MOVIE“ mit sich selbst. Da in der Tabelle nicht nur Filme, sondern auch Serien sind, muss man eine Verbindung zwischen der TV-Serie und seiner Episoden herstellen. Also wurde eine Kante zwischen den Filmen erstellt, mit dem Namen „HAS_EPISODE“ und mit einer die „EPISODEOF“ erstellt, um genau zu bestimmten zu welcher Serie eine Episode gehört. Dabei wurde den Kanten die Properties „episodeyear“, „episode“ und „season“ gegeben.

Zwischen dem Knoten „MOVIE“ und „PERSON“ mussten mehrere Kanten erstellt werden, um die Verschiedenen zusammenhänge zwischen den Knoten zu sein. Da unter den Personen nicht nur Schauspieler oder Producer existieren, sondern auch Leute, die zur Crew gehören.
Dafür habe ich die Tabellen „IMDB_CAST“ und „IMDB_SLV_ROLETYPE“ zusammengefügt um damit die Verbindung zwischen „MOVIE“ und „PERSON“ herzustellen, bei der ich den Rollennamen als Property hinzufüge.

Um eine bessere Übersicht über alle Erstellten Kanten zu haben wurde von mit eine Tabelle angelegt, die in teilen in Abbildung 4.20 und Abbildung 4.21 zu sehen ist, der Rest befindet sich im Anhang.

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:ALT_LANG	none	MOVIE
	:EDITED_FROM	none	
	:EDITED_INTO	none	
	:FEATURED_IN	none	
	:FEATURES	none	
	:FOLLOWED_BY	none	
	:FOLLOWS	none	
	:REFERENCED_IN	none	
	:REFERENCES	none	
	:REMADE_AS	none	
	:REMAKE_OF	none	
	:SPIN_OFF	none	
	:SPIN_OFF_FROM	none	
	:SPOOFED_IN	none	
	:SPOOFS	none	
	:VERSION_OF	none	

Abbildung 4.20: Kantenbeschreibung zwischen „MOVIE“ und „MOVIE“

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:HAS_TITLE	language	AKA_TITLE

Abbildung 4.21: Kantenbeschreibung zwischen „MOVIE“ UND „AKA_TITLE“

5 Realisierung in Neo4j

Nachdem die Modellierung des Graphen fertig war, habe ich mich an die Realisierung in Neo4j gemacht. Dafür habe ich die Enterprise Version von Neo4j genutzt, um den Import der Daten am schnellsten zu gewährleisten. Meine folgenden Schritte werden in Unterkapiteln erklärt. Die Daten müssen jedoch zuerst exportiert werden, als CSV Dateien und dann importiert werden. Der Import der Dateien kann dabei auf verschiedene Arten funktionieren. Es gibt einen CSV Import mit Cypher und auch einen über das Terminal, der schneller funktioniert, aber auch zu Fehlern führen kann.

5.1 Vorbereitung für die Umsetzung

Für die Realisierung der Datenbank mussten die Daten aus der Oracle Datenbank als CSV Datei mit Headern exportiert werden. Dafür musste ich Datenbanken jedoch mit dem „JOIN“ Befehl verbinden um die Daten wie in der Modellierung Importieren zu können.
Dabei führt man die Befehle in Oracle SQL Developer aus und klickt mit der Rechten Maus auf das Abfrageergebnis und klickt auf Exportieren. Danach wählt man das gewünschte Format an, bei mir CSV, wählt die Checkbox „Überschrift“ an und danach den Speicherort.

Dabei habe ich die Daten mehrmals falsch exportiert durch Denkfehler. Mir wäre es ein leichtes Gewesen die Umsetzung durch ein ETL Tool durchzuführen, jedoch wäre das nicht befriedigend für mich gewesen. Der Import hat durch die große Menge an Daten viel Zeit gekostet. Ein paar meiner Aufrufe, um die Daten für den Export vorzubereiten stelle ich nun vor, der Rest wird von mir in den Anhang gepackt.

```
SELECT NVL(RTRIM(LTRIM(regexp_substr(j.NOTE, '([^()]+)'), ') '),')'), j.NOTE) AS LANGUAGE,
j.MOVIE_AKA_TITLE AS MOVIE_AKA_TITLE,
j.MOVIE_ID AS MOVIE_ID,
j.ID AS TITLE_ID
FROM IMDB.IMDB_AKA_TITLES j;
```

Abbildung 5.1: Export der Entität „AKA_TITLE“

Der erste Export wird in Abbildung 5.1 gezeigt.
Da die Sprache in dem der Title vorhanden ist mit Klammern versehen war habe ich eine Kombination von den Befehlen „NVL“, „RTRIM“, „LTRIM“ und „REGEXP“ benutzt. „NVL“ sorgt dafür das Nullwerte mit einem String ersetzt werden (Q:Oracle Webseite). „RTRIM“ und „LTRIM“ schneiden jeweils Links und Rechts bestimmte Zeichen eines Stringes ab und mit „REGEXP“ kann man sich einen String innerhalb

von bestimmten Chars ausgeben lassen. Ich musste „LTRIM“ und „RTRIM“ benutzen, da der String mit „REGEXP“ mit Klammern ausgegeben wurden. Beide Befehle konnte ich jedoch nicht ohne „REGEXP“ benutzen, da es vorkommen konnte, das Extra Informationen Hinter den Klammern erscheinen konnten. Der Rest des Befehls ist ein Einfaches „SELECT FROM“ in dem ich bestimmte Spaltennamen mit den Knotennamen gleichgesetzt habe. Da viele Exports auch für die Kanten benötigt wurden, habe ich die dazu gehörigen IDs mit Ausgegeben.

Der Nächste Export zeigt wie der Knoten „PERSON“ mit seinen Kanten für „CAST_TYPE“ erstellt wurden. (Abbildung 5.2)

```
SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
FROM (IMDB_CAST p
INNER JOIN IMDB_PERSON e
ON p.PERSON_ID = e.ID)
INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
ON p.ROLETYPE_ID = b.ROLETYPE_ID
WHERE b.ROLETYPE_TEXT = 'actor';
```

Abbildung 5.2: Export der Entität „CAST_TYPE“

Hier habe ich mit dem Befehl „INNER JOIN“ benutzt und die Tabellen „IMDB_PERSON“ mit den Tabellen „IMDB_SLV_ROLETYPE“ und „IMDB_CAST“ zusammengeführt. Hierbei habe ich alle Typen die in der Spalte „MOVIELINK_TYPE_TEXT“ aus „IMDB_SLV_ROLETYPE“ nacheinander durch eine „WHERE“ Abfrage ausgegeben und mir diese in eine CSV Datei exportiert. Bestimmt hätte es eine elegantere Lösung dafür gegeben, so war es für mich jedoch einfacher die Kanten beim Import zu erstellen. Für die Kante „:HAS_GOOFS“ musste ich die Daten mehrfach Exportieren, jedoch konnte ich diese Tabelle nicht Importieren. Durch mehrfach nicht korrekt gesetztes Anführungszeichen innerhalb des Textfeldes, wurde der Import verhindert, da immer die Meldung erscheint ist, dass die „mi_info_id“ nicht NULL sein darf. In Abbildung 5.3 ist der Versuch den Export noch zu reparieren zu sehen.

```
SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI_INFOTYPE,
m.MI_INFOTYPE_TEXT AS INFO_MI
FROM IMDB_MOVIE_INFO p
INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI m
ON p.MI_INFOTYPE_ID = m.MI_INFOTYPE_ID
WHERE m.MI_INFOTYPE_TEXT = 'goofs' AND p.ID IS NOT NULL AND m.MI_INFOTYPE_TEXT IS
NOT NULL AND p.MOVIE_ID IS NOT NULL;
```

Abbildung 5.3: Export der Entität „MI_INFO“ mit „goofs“

Trotz des Versuches mit dem Befehl „IS NOT NULL“ bei der besagten Stelle, tauchte der Fehler weiter auf. Da ich die Umsetzung jedoch nicht durch so einen Fehler verhindern wollte, habe ich diese Tabelle nicht in die Finale Graphdatenbank eingefügt. Leider konnte ich Namen oder Filmtitel mit Sonderzeichen nicht korrekt exportieren. Ich habe beim Export und für die Umgebung eine andere Codierung versucht und

auch versucht mit dem Befehl „CONVERT“ eine richtige Ausgabe für die Zeilen zu Gewährleisten. Leider haben alle meine Versuche nichts gebracht, im Internet gab es nur die Lösung die Datenbank nochmal mit der richtigen Codierung zu erstellen, was mir aber wegen fehlender Rechte nicht möglich.

5.2 Erklärung der Importbefehle in Neo4j mit Cypher

Der Import in Cypher wurde mit einigen Befehlen durchgeführt, welche ich in diesem Kapitel zeige und erkläre. Dafür habe ich allen Knoten vorher einen Index oder ein Constraint geben um den Import, mit den aus meinen Befehlen häufig aufkommenden „MERGE“ Befehl, zu beschleunigen. Einen Index legt man mit Cypher mit dem Befehl aus Abbildung 5.4 an.

```
CREATE INDEX ON :MOVIE(id);
```

Abbildung 5.4: Erstellen eines Index in Cypher.

Und ein Constraint mit dem Befehl aus Abbildung 5.5.

```
CREATE CONSTRAINT ON (:PERSON) ASSERT p.personid IS UNIQUE;
```

Abbildung 5.5: Erstellen eines Constraint in Cypher

Wichtig ist es das diese Befehle vor dem eigentlichen Export erfolgen müssen. Ist dies erledigt, wird der Import durchgeführt. Zuerst wurde der Knoten „MOVIE“ erstellt, da alle Kanten auf ihn zeigen werden. (siehe Abbildung 5.6)

```
IMPORT KNOTEN MOVIE

USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE.csv" AS csvLine
CREATE(m:MOVIE {id: toInt(csvLine.MOVIE_ID), fulltitle: csvLine.FULLTITLE, title:
csvLine.TITLE, year: toInt(csvLine.YEAR), kind: csvLine.KIND});
```

Abbildung 5.6: CSV Import per Cypher des Knoten „MOVIE“

Wie man sieht lese ich die Datei aus meinem lokalem Filesystem hoch und definiere dann die aus Unterabschnitt 4.3.1 festgelegten Properties für den Knoten. Diese ist ein noch recht einfacher Import, da er ohne eine Relation zu einem anderen Knoten erstellt wird.
Nun habe ich den zweit größten Knoten nämlich den „PERSON“-Knoten importiert. (siehe Abbildung 5.7)
Diesen Befehl habe ich danach mit den Unterschiedlichen Knoten Labels wiederholt. Dabei besaß jede Datei nur eine bestimmte Anzahl an Personen, nämlich die, die nur zum Beispiel zum Roletype „Actor“ gehören. Bei diesem Befehl habe ich den „MERGE“ Befehl zum anlegen von Personen benutzt, um zu verhindern das keine Doppelten

```
USING PERIODIC COMMIT 500
LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_ACTRESS.csv" AS csvLine
MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});

USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_ACTRESS.csv" AS csvLine
MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
toInt(csvLine.MOVIEID)})
CREATE(person)-[:ACTED_IN { role: csvLine.ROLENAME }]->(movie);
```

Abbildung 5.7: CSV Import per Cypher des Knoten „PERSON“ mit Relation „ACTED_IN“.

Personen vorhanden sind. Danach habe ich die Kante mit dem aus Kapitel 4.3.2 festgelegten Properties angelegt.
Referenziert ein Knoten sich selbst, wie dies bei der Kante „:FOLLOWED_BY“ passiert, sieht der Befehl zum Erstellen ist in Abbildung 5.8 abgebildet.

```
USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_FOLLOWED_BY.csv" AS csvLine
MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
toInt(csvLine.DEST_MOVIEID)})
CREATE(movie)-[:FOLLOWED_BY]->(moviedest)
```

Abbildung 5.8: CSV Import per Cypher der Kante „:FOLLOWED_BY“.

Dabei nehme ich die aus dem CSV Export entstandene Spalte „SRC_MOVIE_ID“ und die Spalte „DEST_MOVIE_ID“. Durch „MERGE“ wird überprüft ob die jeweiligen IDs schon existieren und eine dann eine Verbindung zwischen den beiden erstellt. Auch hier habe ich die CSV Dateien einzeln nach ihrem „MOVIELINK_TYPE_TEXT“ exportiert, um die Kanten danach besser herzustellen.
In meinem nächsten Beispiel erweitere ich den Knoten „MOVIE“ mit dem Befehl „SET“ um einige Properties. (siehe Abbildung 5.9)

```
USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM "file:///DBNew/MPAA_RATING.csv" AS csvLine
MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)})
SET movie.mpaa_rating = csvLine.RATED_TEXT;
```

Abbildung 5.9: CSV Import per Cypher mit „SET“.

Hier arbeite ich allein mit dem „MATCH“ Befehl, dieser findet die jeweiligen Knoten und fügt ihnen dann mit „SET“ die jeweiligen Properties hinzu. Diese Methode habe ich benutzt, da der Export von allen Properties mit denen ich den Knoten „MOVIES“ füllen wollte, nicht funktioniert hat. Jeder Film wurde mehrfach angezeigt ohne einen mir sichtlichen Unterschied. Natürlich hätte man mit „SELECT DISTINCT“ arbeiten können, aber ich hatte Angst das dadurch nicht alle Informationen angezeigt werden können. Deshalb wurden die benötigten Properties erst später eingefügt.
Als letzten Beispiel Import Befehl habe ich einen Befehl mit einer „FOREACH“ Schleife benutzt. Diese wird nicht offiziell in der Dokumentation für den Import erwähnt, hat sich aber in der Praxis bei der Benutzung als Hilfreich erwiesen. (siehe Abbildung 5.10)

```
ERSTELLEN RELATIONS VON SERIEN UND EPISODEN

USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_EPISODE.csv" AS csvLine
MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(episodeof:MOVIE { id:
toInt(csvLine.EPISODEOF)})
FOREACH(ignoreMe IN CASE WHEN csvLine.SEASON <> "" OR csvLine.EPISODE <> "" THEN [1]
ELSE [0] END |
CREATE(movie)-[:EPISODEOF { episode: csvLine.EPISODE, season: csvLine.SEASON,
episodeyear: csvLine.EPISODESYEAR}]->(episodeof)-[:HAS_EPISODE { episode:
csvLine.EPISODE, season: csvLine.SEASON, episodeyear: csvLine.EPISODESYEAR}]->(movie))
```

Abbildung 5.10: CSV Import per Cypher mit „FOREACH“.

Hier wird die „FOREACH“ Schleife dafür benutzt, um zu überprüfen ob die Spalten „SEASON“ oder „EPISODE“ einen NULL Wert besitzen. Dazu wird der „CASE“ Befehl benutzt um den Fall das NULL-Werte erscheinen diese abzufangen. Ist dies der Fall werden die Kanten zwischen „MOVIES“ und „MOVIES“ ohne die Properties erstellt. Haben die Zeilen jedoch einen Wert, werden die Kanten mit den Properties erstellt. So habe ich keine leeren Werte in den Properties stehen und habe trotzdem die richtige Verbindung zwischen den Knoten hergestellt.
Die anderen Import Befehle sind im Anhang zu finden.

5.3 Erklärung der Möglichen Datenabfrage

Nun da Graph in Neo4j erstellt wurde, können wir Abfragen erstellen, um uns bestimmte Pfade oder auch Knoten anzeigen zu lassen. Da die Datenbank nach dem Import fast 14 GB beträgt, können die Daten immer nur begrenzt angezeigt zu werden. Dazu kann man mit dem Befehl „LIMIT“ die Ausgabe limitieren. Die Abfrage mit einem Limit könnte wie in Abbildung 5.11 aussehen.

```
$ MATCH p=(:ALIAS)--(:PERSON) RETURN p LIMIT 50
```

Abbildung 5.11: Cypher Abfrage mit „LIMIT“.

Die Ausgabe kann man in Abbildung 5.12 sehen.

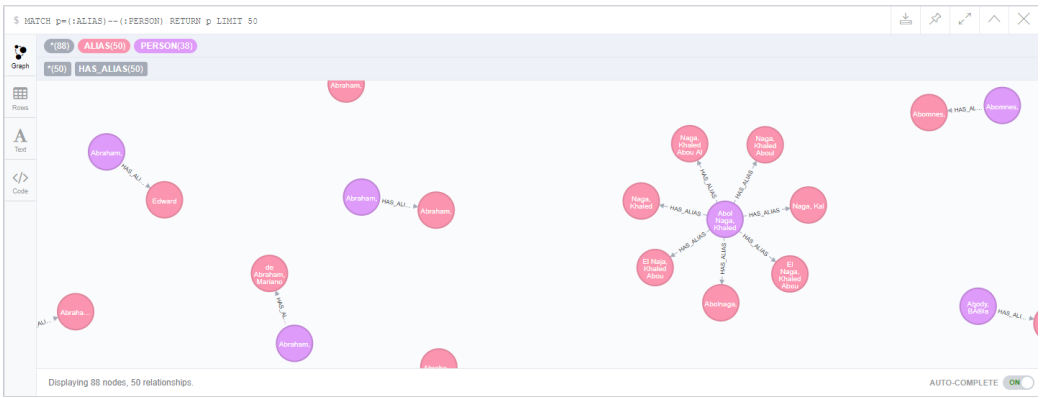

Abbildung 5.12: Cypher Ausgabe mit „LIMIT“.

Wie wir sehen, bekommen wir die ersten 50 Graphen zwischen dem Knoten „ALIAS“ und „PERSON“ ausgegeben. Wir haben bei der Ausgabe verschiedene Darstellungsmöglichkeiten, als Tabelle, als Graph (Standard), als Text und als Code. Möchte man die Properties der Knoten und Kanten in der Graph Ansicht sehen, hovert man über den gewünschten Knoten oder Kante und die Properties werden unter der Anzeige der Graphen angezeigt. Sollte dies einem zu Umständlich sein, kann dies auch über die „Rows“ Ansicht geschehen.
Eine weitere Datenabfrage könnte mit „WHERE“ geschehen, ein Beispiel dazu zeigt Abbildung 5.13.

```
$ MATCH (n:MOVIE) WHERE n.title=~'.*Fight Club.*' RETURN n
```

Abbildung 5.13: Cypher Abfrage mit „WHERE“.

Die Abfrage funktioniert dabei wie in SQL, wir wollen vom Label „MOVIE“ die Knoten angezeigt bekommen deren Propertie „name“ gleich „Fight Club“ sind.
Unsere Ausgabe würde bei unserem Graphen wie in Abbildung 5.14 aussehen.

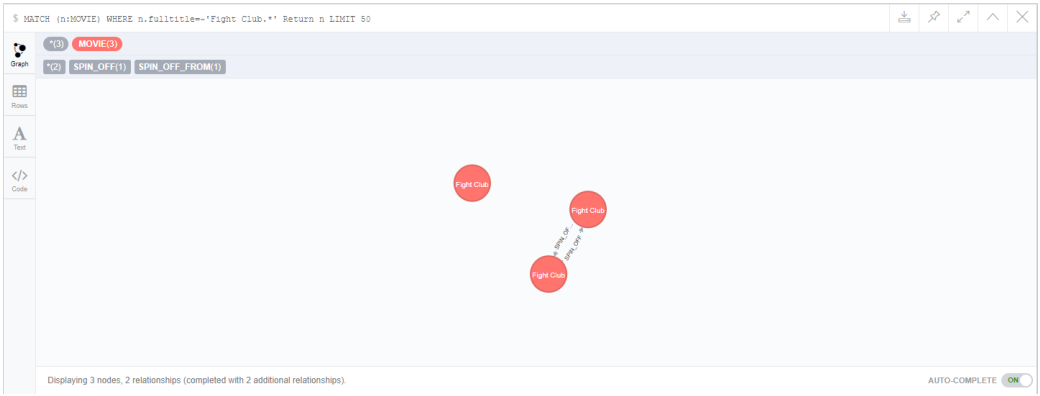

Abbildung 5.14: Cypher Ausgabe mit „WHERE“.

Wenn wir uns alle Personen die in dem Film „Fight Club“ mitgespielt haben anzeigen wollen benutzen wir den Aufruf aus Abbildung 5.15.

```
$ MATCH p=(:PERSON)-[:ACTED_IN]->(m:MOVIE) WHERE m.fulltitle=~'Fight Club.*' RETURN p
```

Abbildung 5.15: Cypher Abfrage mit „WHERE“ und „LIMIT“.

Damit zeigen wir uns alle Personen an die im Film „Fight Club“ mitgespielt haben. Die Ausgabe sieht dabei wie in Abbildung 5.16 aus.

Wie man bemerkt hat ist die Abfrage von Befehlen sehr leicht verständlich und die Abfragen schnell erstellt oder auch bearbeitet. Da jede Abfrage durch seine Notation

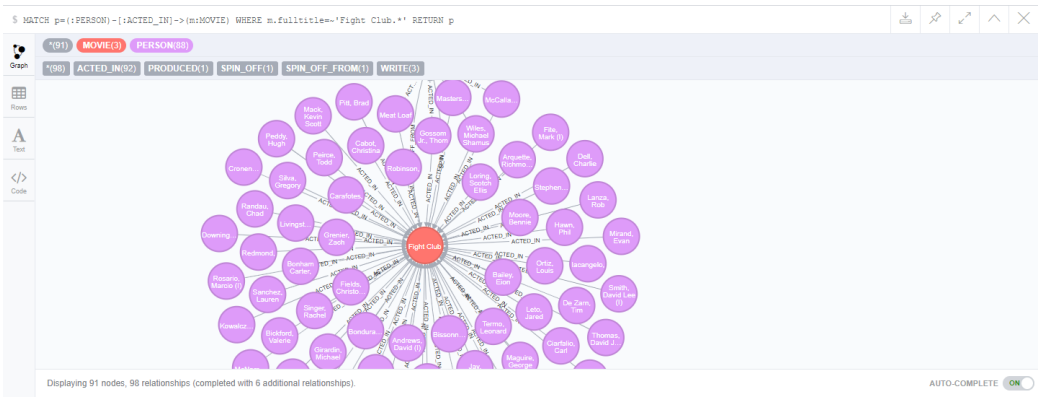

Abbildung 5.16: Cypher Ausgabe mit „WHERE“ und „LIMIT“.

wie ein Graph aussieht, ist die Einfachheit der Abfragen nicht zu übersehen.
Dadurch ist es auch möglich größere Abfragen zu Gestalten um mehr Informationen anzuzeigen.

```
$ MATCH p=(:BIO)--(:PERSON)-[:ACTED_IN]->(m:MOVIE) WHERE m.fulltitle=~'Fight Club.*' RETURN p
```

Abbildung 5.17: Cypher größere Abfrage.

In Abbildung 5.17 lassen wir uns auch noch die Informationen aus den Knoten „BIO“ für die Schauspieler ausgeben die im „Fight Club“ mitgespielt haben.

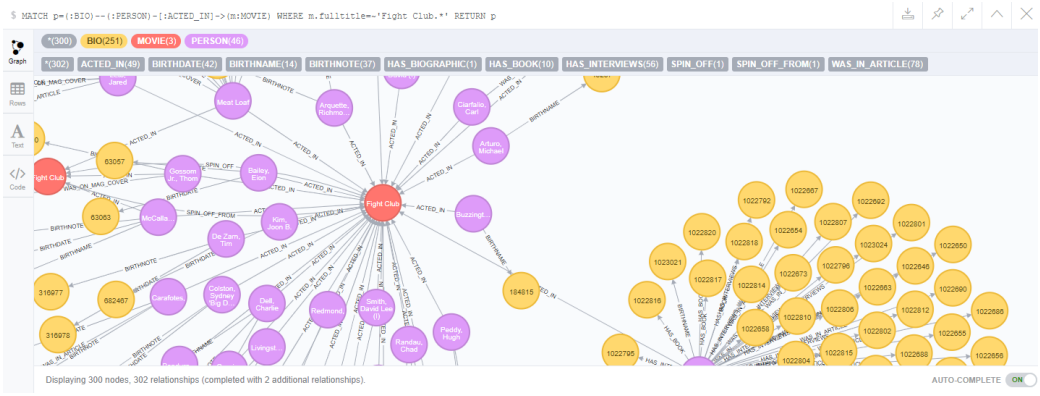

Abbildung 5.18: Cypher größere Abfrage..

Wir könnten uns nun auch noch weitere Informationen wie den Plot oder die Einspielergebnisse des Films ausgeben lassen.
Es ist durch die Einfachheit von Cypher möglich leicht Abfragen für bestimmte Daten zu erstellen. Auch das löschen von Properties, Labels, Kanten und Knoten ist damit möglich. Wollen wir den Namen für den Schauspieler „Keanu Reeves“ ändern, können wir dies mit dem Befehl aus Abbildung 5.19 zeigen.

```
$ MATCH(p:PERSON {name: 'Reeves, Keanu'}) SET p.name='Kinu Reeves' RETURN p
```

Abbildung 5.19: Cypher größere Abfrage.

Sollte man sich den ganzen Graphen mit der Kante „:ACTED_IN“ ausgeben lassen wollen, sollte dies lieber mit einem Graph Visualization Tool passieren, welches man einfach in Neo4j einfügen kann. Ohne ein solches Tool, kann die Abfrage ohne entsprechenden Index auf die Properties sehr lange dauern, da die Visualisierung des Graphen sehr viel Rechnerleistung kostet.

6 Fazit und Ausblick

Zum Ende hin musste ich feststellen, dass Neo4j eine gute Plattform für Datenbanken ist, aber man noch merkt, dass ihre Plattform laufend bearbeitet wird. Viele Fehler die ich mit der Datenbank hatte, konnten mit im Internet nicht beantwortet werden. Ich musste häufiger die Datenbank neu aufsetzen damit ich eine laufende Datenbank als Ergebnis abgeben konnte.
Auch habe ich mich in meinem Zeitplan verkalkuliert und die Zeit, die die Realisierung der Datenbank gebraucht hat, unterschätzt. Ich denke auch das meine Umsetzung noch nicht perfekt ist und ich noch viel an Arbeit reinstecken kann.
Die Verbindungen zwischen „MOVIE“ und „MOVIE“ über „MOVIE_LINKS“ enthält noch zu viele doppelte Kanten die Eigendlich das gleiche über die Verbindung aussagt. Man könnte auch viele Informationen als Liste oder Set direkt „MOVIE“ übergeben und so Knoten und Relationen sparen.
Ob die Benutzung von IDs waren nicht immer notwendig, haben mir aber geholfen, nicht den Überblick bei der Verbindung der Knoten zu verlieren.

Am Ende bin ich trotzdem zufrieden mit meiner Arbeit, da die Umsetzung doch schwerer war als gedacht und ich dies nicht einem ETL-Tool überlassen habe. Dies wäre zum Ende der einfachere Weg gewesen um einen Graphen aus einer Relationalen Datenbank zu machen. Auch hat mir die Arbeit mit Neo4j viel Spaß gemacht und mir gezeigt, was für tolle Alternativen es in diesem Bereich gibt und ich würde gerne weitere NoSQL-Datenbanken kennen lernen.
Eine Umsetzung eines weiteren Projektes in Neo4j würde ich daher nicht ausschließen, da es noch viel für mich zu entdecken gibt. Ich denke bisher habe ich nur an der Oberfläche gekratzt und das noch mehr Power in der Datenbank steckt.
Alleine die vielen Plugins, die Neo4j noch mächtiger und für Entwickler angenehmer machen, haben mein Interesse geweckt. Auch die gute Anbindung für neue Technologien wie GraphQL sind Zeichen dafür, das Neo4j eine Datenbank der Zukunft ist. Big-Data Projekte oder Projekte mit großem Such aufkommen, sollten Neo4j oder Graphendatenbanken in Betracht ziehen zu benutzen.
Ich gehe aus diesem Projekt mit einem Positiven Gefühl raus und habe viele dabei an Erfahrung mitgenommen.

Abbildungsverzeichnis

Literaturverzeichnis

[Bryce a] Bryce, Merkl S.: *Graph Databases for Beginners: Data Modeling Pitfalls to Avoid*. https://neo4j.com/blog/data-modeling-pitfalls/. – Last accessed 15 Januar 2019

[Bryce b] Bryce, Merkl S.: *Graph Databases for Beginners: Other Graph Technologies*. https://neo4j.com/blog/other-graph-database-technologies/. – Last accessed 15 Januar 2019

[Destatis] Destatis: *Destatis*. https://www-genesis.destatis.de. – Last accessed 15 Januar 2019

[Edlich 2010] Edlich, Friedland Achim Hampe Jens Brauer B. Stefan: *NoSQL - Einstieg in die Welt Nichtrelationaler WEB 2.0 Datenbanken*. 1. Auflage. München : Carl Hanser Verlag, 2010. – ISBN 978-3-446-42355-8

[Hunger 2014] Hunger, Michael: *Neo4j 2.0 - Eine Graphdatenbank für alle*. 1. Auflage. entwickler.press, 2014. – ISBN 9783-868802-315-2

[Jarosch 2010] Jarosch, Helmut: *Grundkurs Datenbankentwurf*. 3. Auflage. GWV Fachverlage GmbH, Wiesbaden : Vieweg+Teubner, 2010. – ISBN 978-3-8348-0955-1

[Neo4jInc a] Neo4jInc: *Graph Modeling Guidelines*. https://neo4j.com/developer/guide-data-modeling/. – Last accessed 15 Januar 2019

[Neo4jInc b] Neo4jInc: *The Neo4j Cypher Manual v3.5*. https://neo4j.com/docs/cypher-manual/3.5/. – Last accessed 15 Januar 2019

[Neo4jInc c] Neo4jInc: *Neo4j Graph Platform*. https://neo4j.com/developer/graph-platform/. – Last accessed 15 Januar 2019

[Neo4jInc d] Neo4jInc: *Subscription*. https://neo4j.com/subscriptions/. – Last accessed 15 Januar 2019

[Robinson 2015] Robinson, Webber Jim Eifrem E. Ian: *Graph Database*. 2. Auflage. 1005 Gravenstein Highway North, Sebastiopol : O'Reilly, 2015. – ISBN 978-1-491-93200-1

[Yu] Yu, Fanghua: *Data Profiling: A Holistic View of Data using Neo4j*. https://neo4j.com/blog/data-profiling-holistic-view-neo4j/. – Last accessed 15 Januar 2019

Anhang

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:ALT_LANG	none	MOVIE
	:EDITED_FROM	none	
	:EDITED_INTO	none	
	:FEATURED_IN	none	
	:FEATURES	none	
	:FOLLOWED_BY	none	
	:FOLLOWS	none	
	:REFERENCED_IN	none	
	:REFERENCES	none	
	:REMADE_AS	none	
	:REMAKE_OF	none	
	:SPIN_OFF	none	
	:SPIN_OFF_FROM	none	
	:SPOOFED_IN	none	
	:SPOOFS	none	
	:VERSION_OF	none	

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:ALTERNATIVE_VERSIONS	none	MI_INFO
	:HAS_CRAZY_CREDITS	none	
	:HAS_QUOTES	none	
	:SOUNDTRACK	none	
	:HAS_TRIVIA	none	

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:HAS_CERTIFICATE	none	MI2_INFO
	:HAS_COLOR_INFO	none	
	:COUNTRY	none	
	:HAS_DISTRIBUTORS	none	
	:FROM_GENRE	none	
	:HAS_KEYWORDS	none	
	:EXIST_IN_LANG	none	
	:HAS_LOCATIONS	none	
	:WITH_MISC_COMPANIES	none	
	:WITH_PRODUCTION_COMPANIES	none	
	:RELEASE_DATES	none	
	:RUNTIME	none	
	:SOUND_MIX	none	
	:SPECIAL_EFFECTS_MADE_BY	none	
	:TECH_INFO	none	

Abbildung 6.1: Tabelle Kanten

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:COST_ADMISSION	none	BUSINESS
	:HAD_BUDGET	none	
	:COPYRIGHT_HOLDER	none	
	:WAS_FILMED	none	
	:PLAYED_GROSS	none	
	:PLAYED_ON_OPENING_WEEKEND	none	
	:PRODUCTED	none	
	:RENTALS	none	
	:GROSS_PLAYED_ON_OPENING_WEEKEND	none	

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:HAS_TITLE	language	AKA_TITLE

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:HAS_PLOT	none	PLOT

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE	:HAS_EPISODE	none	MOVIE:EPISODE

Startknoten	Kantenlabel	Properties	Endknoten
MOVIE:EPISODE	:EPISODEOF	episode, season, episodeofyear	MOVIE

Startknoten	Kante	Property	Endknoten
PERSON	:HAS_AGENT	none	BIO
	:WAS_IN_ARTICLE	none	
	:HAS_BIOGRAPHIC	none	
	:BIRTHDATE	none	
	:BIRTHNAME	none	
	:BIRTHNOTE	none	
	:HAS_BOOK	none	
	:DEATH_DATE	none	
	:DEATH_NOTE	none	
	:HAS_HEIGHT	none	
	:HAS_INTERVIEWS	none	
	:WAS_ON_MAG_COVERS	none	

Startknoten	Kantenlabel	Properties	Endknoten
PERSON	:HAS_ALIAS	none	ALIAS
	:HAS_NICK_NAME	none	
	:HAS_BIRTH_NAME	none	

Abbildung 6.2: Tabelle Kanten

Startknoten	Kantenlabel	Properties	Endknoten
PERSON	:ACTED_IN	role	MOVIE
	:CINEMATOGRAPH	role	
	:COMPOSE	role	
	:DESIGN_COSTUME	role	
	:DIRECTED	role	
	:EDITED	role	
	:WRITE	role	
	:DESIGN_PRODUCTION	role	
	:PRODUCED	role	
	:MISC_CREW	role	

Abbildung 6.3: Tabelle Kanten

AKA_TITLE

Propertyname	Property Beschreibung	Datentyp
titleid	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, weil der Titel nicht eindeutig genug wäre.	Integer
title	Der Alternative Titel des Filmes.	String

ALIAS

Propertyname	Property Beschreibung	Datentyp
aliasid	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, weil der Alias nicht immer eindeutig war.	Integer
name	Der Alias, Geburtsname oder Nick Name der Person.	String

BIO

Propertyname	Property Beschreibung	Datentyp
bioid	Enthält die ID zum Identifizieren des Knoten, da die Bio häufig vom Text zu lang für die ID wäre.	Integer
text	Enthält den Bio Text.	String

BUSINESS

Propertyname	Property Beschreibung	Datentyp
businessid	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, um eine gute Verbindung herzustellen, da es verschiedene Kanten für BUSINESS gibt.	Integer
businesstext	Der Inhalt des Knotens.	String

MI_INFO

Propertyname	Property Beschreibung	Datentyp
mi_info_id	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, da es verschiedene Kanten für MI_INFO gibt.	Integer
Info	Enthält den Infotext.	String

MI2_INFO

Propertyname	Property Beschreibung	Datentyp
mi2_info_id	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, da es verschiedene Kanten für MI2_INFO gibt.	Integer
info	Enthält den Infotext	String

Abbildung 6.4: Tabelle Knoten

MOVIE

Propertyname	Property Beschreibung	Datentyp
id	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, weil der Titel nicht eindeutig genug wäre und ich das Problem mit den Sonderzeichen nicht lösen konnte.	Integer
fulltitle	Der volle Titel des Films.	String
title	Dieser Titel enthält einen kurzen Titel.	String
year	Enthält das Jahr.	String
kind	Enthält die Art des Filmes.	String
tagline	Ist die Tagline des Filmes.	String
rating	Gibt das momentane Rating des Filmes an.	String
votes	Die Votes an die abgegeben wurden.	String
votes_distribution	Gibt die Verteilung der Votes an.	String
mpaa_rating	Ist die Alterseinstufung des Filmes.	String

PERSON

Propertyname	Property Beschreibung	Datentyp
personid	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, weil der der Name der Person mit Leerzeichen bei mir zu Fehlern geführt hat beim Erstellen der Kanten.	Integer
name	Enthält den Namen der Person.	String

PLOT

Propertyname	Property Beschreibung	Datentyp
plotid	Enthält die ID zum Identifizieren des Knoten, wurde gewählt, weil der Infotext zu lang gewesen wäre zum Identifizieren.	Integer
info	Der Plot des Filmes.	String
author	Wer den Plot auf der Seite geschrieben hat.	String

Abbildung 6.5: Tabelle Knoten

```
1
2   IMPORT KNOTEN MOVIE
3
4   USING PERIODIC COMMIT 500
5   LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE.csv" AS csvLine
6   CREATE(m:MOVIE {id: toInt(csvLine.MOVIE_ID), fulltitle: csvLine.FULLTITLE, title:
    csvLine.TITLE, year: toInt(csvLine.YEAR), kind: csvLine.KIND});
7
8   CREATE INDEX ON :MOVIE(id);
9
10  CREATE CONSTRAINT ON (:PERSON) ASSERT p.personid IS UNIQUE;
11
12
13  IMPORT KNOTEN PERSON MIT RELATIONS
14
15  USING PERIODIC COMMIT 500
16  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_ACTOR.csv" AS csvLine
17  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
18
19  USING PERIODIC COMMIT 5000
20  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_ACTOR.csv" AS csvLine
21  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
22  CREATE(person)-[:ACTED_IN { role: csvLine.ROLENAME }]->(movie);
23
24  USING PERIODIC COMMIT 500
25  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_ACTRESS.csv" AS csvLine
26  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
27
28  USING PERIODIC COMMIT 5000
29  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_ACTRESS.csv" AS csvLine
30  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
31  CREATE(person)-[:ACTED_IN { role: csvLine.ROLENAME }]->(movie);
32
33  USING PERIODIC COMMIT 500
34  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_CINEMATOGRAPH.csv" AS csvLine
35  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
36
37  USING PERIODIC COMMIT 5000
38  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_CINEMATOGRAPH.csv" AS csvLine
39  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
40  CREATE(person)-[:CINEMATOGRAPH { role: csvLine.ROLENAME }]->(movie);
41
42  USING PERIODIC COMMIT 500
43  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_COMPOSER.csv" AS csvLine
44  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
45
46  USING PERIODIC COMMIT 5000
47  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_COMPOSER.csv" AS csvLine
48  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
49  CREATE(person)-[:COMPOSE { role: csvLine.ROLENAME }]->(movie);
50
51  USING PERIODIC COMMIT 500
52  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_COSTUME_DESIGNER.csv" AS csvLine
53  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
54
55  USING PERIODIC COMMIT 5000
56  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_COSTUME_DESIGNER.csv" AS csvLine
57  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
58  CREATE(person)-[:DESIGN_COSTUME { role: csvLine.ROLENAME }]->(movie);
59
60  USING PERIODIC COMMIT 500
61  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_DIRECTOR.csv" AS csvLine
62  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
63
64  USING PERIODIC COMMIT 5000
65  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_DIRECTOR.csv" AS csvLine
66  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
```

Abbildung 6.6: Import mit Cypher

```
67  CREATE(person)-[:DIRECTED { role: csvLine.ROLENAME }]->(movie);
68
69  USING PERIODIC COMMIT 500
70  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_EDITOR.csv" AS csvLine
71  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
72
73  USING PERIODIC COMMIT 5000
74  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_EDITOR.csv" AS csvLine
75  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
76  CREATE(person)-[:EDITED { role: csvLine.ROLENAME }]->(movie);
77
78  USING PERIODIC COMMIT 500
79  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_MISCELLANEOUSCREW.csv" AS
    csvLine
80  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
81
82  USING PERIODIC COMMIT 5000
83  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_MISCELLANEOUSCREW.csv" AS
    csvLine
84  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
85  CREATE(person)-[:MISC_CREW { role: csvLine.ROLENAME }]->(movie);
86
87  USING PERIODIC COMMIT 500
88  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_PRODUCER.csv" AS csvLine
89  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
90
91  USING PERIODIC COMMIT 5000
92  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_PRODUCER.csv" AS csvLine
93  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
94  CREATE(person)-[:PRODUCED { role: csvLine.ROLENAME }]->(movie);
95
96  USING PERIODIC COMMIT 500
97  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_PRODUCTION_DESIGNER.csv" AS
    csvLine
98  MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
99
100 USING PERIODIC COMMIT 5000
101 LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_PRODUCTION_DESIGNER.csv" AS
    csvLine
102 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
103 CREATE(person)-[:DESIGN_PRODUCTION { role: csvLine.ROLENAME }]->(movie);
104
105 USING PERIODIC COMMIT 500
106 LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_WRITER.csv" AS csvLine
107 MERGE(p:PERSON {personid: toInt(csvLine.PERSON_ID), name: csvLine.FULLNAME});
108
109 USING PERIODIC COMMIT 5000
110 LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_ROLE_WRITER.csv" AS csvLine
111 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(movie:MOVIE { id:
    toInt(csvLine.MOVIEID)})
112 CREATE(person)-[:WRITE { role: csvLine.ROLENAME }]->(movie);
113
114
115
116
117
118
119
120
121
122
123
124
125 IMPORT ALIAS MIT RELATIONS
126
127 USING PERIODIC COMMIT 500
128 LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_AKA_AKA.csv" AS csvLine
129 MERGE(p:ALIAS {aliasid: toInt(csvLine.AKA_ID), name: csvLine.PERSON_AKA_NAME});
130
```

Abbildung 6.7: Import mit Cypher

```
131  USING PERIODIC COMMIT 5000
132  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_AKA_AKA.csv" AS csvLine
133  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(alias:ALIAS { aliasid:
     toInt(csvLine.AKA_ID)})
134  CREATE(person)-[:HAS_ALIAS]->(alias);
135
136  USING PERIODIC COMMIT 500
137  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_BIRTH_NAME.csv" AS csvLine
138  MERGE(p:ALIAS {aliasid: toInt(csvLine.AKA_ID), name: csvLine.PERSON_AKA_NAME});
139
140  USING PERIODIC COMMIT 5000
141  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_BIRTH_NAME.csv" AS csvLine
142  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(alias:ALIAS { aliasid:
     toInt(csvLine.AKA_ID)})
143  CREATE(person)-[:HAS_BIRTHNAME]->(alias);
144
145  USING PERIODIC COMMIT 500
146  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_NICK_NAME.csv" AS csvLine
147  MERGE(p:ALIAS {aliasid: toInt(csvLine.AKA_ID), name: csvLine.PERSON_AKA_NAME});
148
149  USING PERIODIC COMMIT 5000
150  LOAD CSV WITH HEADERS FROM "file:///DBNew/PERSON_NICK_NAME.csv" AS csvLine
151  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(alias:ALIAS { aliasid:
     toInt(csvLine.AKA_ID)})
152  CREATE(person)-[:HAS_NICK_NAME]->(alias);
153
154
155
156
157
158  IMPORT BIO MIT RELATIONS
159
160  USING PERIODIC COMMIT 500
161  LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_AGENT.csv" AS csvLine
162  MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
163
164  USING PERIODIC COMMIT 5000
165  LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_AGENT.csv" AS csvLine
166  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
     toInt(csvLine.BIO_ID)})
167  CREATE(person)-[:HAS_AGENT]->(bio);
168
169  USING PERIODIC COMMIT 500
170  LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_ARTICLES.csv" AS csvLine
171  MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
172
173  USING PERIODIC COMMIT 5000
174  LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_ARTICLES.csv" AS csvLine
175  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
     toInt(csvLine.BIO_ID)})
176  CREATE(person)-[:WAS_IN_ARTICLE]->(bio);
177
178  USING PERIODIC COMMIT 500
179  LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BIOGRAPHICAL.csv" AS csvLine
180  MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
181
182  USING PERIODIC COMMIT 5000
183  LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BIOGRAPHICAL.csv" AS csvLine
184  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
     toInt(csvLine.BIO_ID)})
185  CREATE(person)-[:HAS_BIOGRAPHIC]->(bio);
186
187  USING PERIODIC COMMIT 500
188  LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BIRTHDATE.csv" AS csvLine
189  MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
190
191  USING PERIODIC COMMIT 5000
192  LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BIRTHDATE.csv" AS csvLine
193  MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
     toInt(csvLine.BIO_ID)})
194  CREATE(person)-[:BIRTHDATE]->(bio);
195
196  USING PERIODIC COMMIT 500
```

Abbildung 6.8: Import mit Cypher

```
197 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BIRTHNAME.csv" AS csvLine
198 MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
199
200 USING PERIODIC COMMIT 5000
201 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BIRTHNAME.csv" AS csvLine
202 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
    toInt(csvLine.BIO_ID)})
203 CREATE(person)-[:BIRTHNAME]->(bio);
204
205 USING PERIODIC COMMIT 500
206 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BIRTHNOTE.csv" AS csvLine
207 MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
208
209 USING PERIODIC COMMIT 5000
210 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BIRTHNOTE.csv" AS csvLine
211 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
    toInt(csvLine.BIO_ID)})
212 CREATE(person)-[:BIRTHNOTE]->(bio);
213
214 USING PERIODIC COMMIT 500
215 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BOOKS.csv" AS csvLine
216 MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
217
218 USING PERIODIC COMMIT 5000
219 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_BOOKS.csv" AS csvLine
220 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
    toInt(csvLine.BIO_ID)})
221 CREATE(person)-[:HAS_BOOK]->(bio);
222
223 USING PERIODIC COMMIT 500
224 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_DEATH_DATE.csv" AS csvLine
225 MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
226
227 USING PERIODIC COMMIT 5000
228 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_DEATH_DATE.csv" AS csvLine
229 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
    toInt(csvLine.BIO_ID)})
230 CREATE(person)-[:DEATH_DATE]->(bio);
231
232 USING PERIODIC COMMIT 500
233 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_DEATH_NOTE.csv" AS csvLine
234 MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
235
236 USING PERIODIC COMMIT 5000
237 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_DEATH_NOTE.csv" AS csvLine
238 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
    toInt(csvLine.BIO_ID)})
239 CREATE(person)-[:DEATH_NOTE]->(bio);
240
241 USING PERIODIC COMMIT 500
242 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_INTERVIEWS.csv" AS csvLine
243 MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
244
245 USING PERIODIC COMMIT 5000
246 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_INTERVIEWS.csv" AS csvLine
247 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
    toInt(csvLine.BIO_ID)})
248 CREATE(person)-[:HAS_INTERVIEWS]->(bio);
249
250 USING PERIODIC COMMIT 500
251 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_MAGAZINE_COVERS.csv" AS csvLine
252 MERGE(p:BIO {bioid: toInt(csvLine.BIO_ID), text: csvLine.BIO_TEXT});
253
254 USING PERIODIC COMMIT 5000
255 LOAD CSV WITH HEADERS FROM "file:///DBNew/BIO_MAGAZINE_COVERS.csv" AS csvLine
256 MATCH(person:PERSON { personid: toInt(csvLine.PERSON_ID)}),(bio:BIO { bioid:
    toInt(csvLine.BIO_ID)})
257 CREATE(person)-[:WAS_ON_MAG_COVER]->(bio);
258
259
260
261
262
```

Abbildung 6.9: Import mit Cypher

```
263
264
265
266  IMPORT BUSINESS MIT RELATIONS
267
268  USING PERIODIC COMMIT 500
269  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_ADMISSIONS.csv" AS csvLine
270  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
271
272  USING PERIODIC COMMIT 5000
273  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_ADMISSIONS.csv" AS csvLine
274  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
275  CREATE(movie)-[:COST_ADMISSION]->(business);
276
277  USING PERIODIC COMMIT 500
278  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_BUDGET.csv" AS csvLine
279  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
280
281  USING PERIODIC COMMIT 5000
282  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_BUDGET.csv" AS csvLine
283  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
284  CREATE(movie)-[:HAD_BUDGET]->(business);
285
286  USING PERIODIC COMMIT 500
287  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_COPYRIGHT_HOLDER.csv" AS csvLine
288  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
289
290  USING PERIODIC COMMIT 5000
291  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_COPYRIGHT_HOLDER.csv" AS csvLine
292  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
293  CREATE(movie)-[:COPYRIGHT_HOLDER]->(business);
294
295  USING PERIODIC COMMIT 500
296  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_FILMING_DATES.csv" AS csvLine
297  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
298
299  USING PERIODIC COMMIT 5000
300  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_FILMING_DATES.csv" AS csvLine
301  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
302  CREATE(movie)-[:WAS_FILMED]->(business);
303
304  USING PERIODIC COMMIT 500
305  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_GROSS.csv" AS csvLine
306  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
307
308  USING PERIODIC COMMIT 5000
309  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_GROSS.csv" AS csvLine
310  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
311  CREATE(movie)-[:PLAYED_GROSS]->(business);
312
313  USING PERIODIC COMMIT 500
314  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_OPENING_WEEKEND.csv" AS csvLine
315  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
316
317  USING PERIODIC COMMIT 5000
318  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_OPENING_WEEKEND.csv" AS csvLine
319  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
320  CREATE(movie)-[:PLAYED_ON_OPENING_WEEKEND]->(business);
321
322  USING PERIODIC COMMIT 500
323  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_PRODUCTION_DATES.csv" AS csvLine
```

Abbildung 6.10: Import mit Cypher

```
324  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
325
326  USING PERIODIC COMMIT 5000
327  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_PRODUCTION_DATES.csv" AS csvLine
328  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
329  CREATE(movie)-[:PRODUCTED]->(business);
330
331  USING PERIODIC COMMIT 500
332  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_RENTALS.csv" AS csvLine
333  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
334
335  USING PERIODIC COMMIT 5000
336  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_RENTALS.csv" AS csvLine
337  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
338  CREATE(movie)-[:RENTALS]->(business);
339
340  USING PERIODIC COMMIT 500
341  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_WEEKEND_GROSS.csv" AS csvLine
342  MERGE(p:BUSINESS {businessid: toInt(csvLine.BUSINESS_ID), businesstext:
     csvLine.BUSINESS_TEXT});
343
344  USING PERIODIC COMMIT 5000
345  LOAD CSV WITH HEADERS FROM "file:///DBNew/BUSINESS_WEEKEND_GROSS.csv" AS csvLine
346  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(business:BUSINESS { businessid:
     toInt(csvLine.BUSINESS_ID)})
347  CREATE(movie)-[:GROSS_PLAYED_ON_OPENING_WEEKEND]->(business);
348
349
350
351
352
353
354  IMPORT KNOTEN MI_INFO MIT RELATIONS
355
356  USING PERIODIC COMMIT 500
357  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_ALTERNATE_VERSION.csv" AS csvLine
358  MERGE(p:MI_INFO {mi_info_id: toInt(csvLine.MI_INFOTYPE), info: csvLine.INFO});
359
360  USING PERIODIC COMMIT 5000
361  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_ALTERNATE_VERSION.csv" AS csvLine
362  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi_info:MI_INFO { mi_info_id:
     toInt(csvLine.MI_INFOTYPE)})
363  CREATE(movie)-[:ALTERNATIVE_VERSIONS]->(mi_info);
364
365  USING PERIODIC COMMIT 500
366  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_CRAZY_CREDITS.csv" AS csvLine
367  MERGE(p:MI_INFO {mi_info_id: toInt(csvLine.MI_INFOTYPE), info: csvLine.INFO});
368
369  USING PERIODIC COMMIT 5000
370  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_CRAZY_CREDITS.csv" AS csvLine
371  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi_info:MI_INFO { mi_info_id:
     toInt(csvLine.MI_INFOTYPE)})
372  CREATE(movie)-[:HAS_CRAZY_CREDITS]->(mi_info);
373
374  USING PERIODIC COMMIT 500
375  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_QUOTES.csv" AS csvLine
376  MERGE(p:MI_INFO {mi_info_id: toInt(csvLine.MI_INFOTYPE), info: csvLine.INFO});
377
378  USING PERIODIC COMMIT 5000
379  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_QUOTES.csv" AS csvLine
380  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi_info:MI_INFO { mi_info_id:
     toInt(csvLine.MI_INFOTYPE)})
381  CREATE(movie)-[:HAS_QUOTES]->(mi_info);
382
383  USING PERIODIC COMMIT 500
384  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_SOUNDTRACK.csv" AS csvLine
385  MERGE(p:MI_INFO {mi_info_id: toInt(csvLine.MI_INFOTYPE), info: csvLine.INFO});
386
387  USING PERIODIC COMMIT 5000
```

Abbildung 6.11: Import mit Cypher

```
388  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_SOUNDTRACK.csv" AS csvLine
389  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi_info:MI_INFO { mi_info_id:
     toInt(csvLine.MI_INFOTYPE)})
390  CREATE(movie)-[:SOUNDTRACK]->(mi_info);
391
392  USING PERIODIC COMMIT 500
393  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_TRIVIA.csv" AS csvLine
394  MERGE(p:MI_INFO {mi_info_id: toInt(csvLine.MI_INFOTYPE), info: csvLine.INFO});
395
396  USING PERIODIC COMMIT 5000
397  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI_TRIVIA.csv" AS csvLine
398  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi_info:MI_INFO { mi_info_id:
     toInt(csvLine.MI_INFOTYPE)})
399  CREATE(movie)-[:HAS_TRIVIA]->(mi_info);
400
401
402
403
404
405
406  IMPORT KNOTEN MI2_INFO
407
408  USING PERIODIC COMMIT 500
409  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_CERTIFICATES.csv" AS csvLine
410  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
411
412  USING PERIODIC COMMIT 5000
413  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_CERTIFICATES.csv" AS csvLine
414  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
415  CREATE(movie)-[:HAS_CERTIFICATE]->(mi2_info);
416
417  USING PERIODIC COMMIT 500
418  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_COLOR.csv" AS csvLine
419  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
420
421  USING PERIODIC COMMIT 5000
422  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_COLOR.csv" AS csvLine
423  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
424  CREATE(movie)-[:HAS_COLOR_INFO]->(mi2_info);
425
426  USING PERIODIC COMMIT 500
427  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_COUNTRIES.csv" AS csvLine
428  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
429
430  USING PERIODIC COMMIT 5000
431  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_COUNTRIES.csv" AS csvLine
432  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
433  CREATE(movie)-[:COUNTRY]->(mi2_info);
434
435  USING PERIODIC COMMIT 500
436  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_DISTRIBUTORS.csv" AS csvLine
437  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
438
439  USING PERIODIC COMMIT 5000
440  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_DISTRIBUTORS.csv" AS csvLine
441  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
442  CREATE(movie)-[:HAS_DISTRIBUTORS]->(mi2_info);
443
444  USING PERIODIC COMMIT 500
445  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_GENRE.csv" AS csvLine
446  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
447
448  USING PERIODIC COMMIT 5000
449  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_GENRE.csv" AS csvLine
450  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
451  CREATE(movie)-[:FROM_GENRE]->(mi2_info);
452
453  USING PERIODIC COMMIT 500
```

Abbildung 6.12: Import mit Cypher

```
454  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_KEYWORDS.csv" AS csvLine
455  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
456
457  USING PERIODIC COMMIT 5000
458  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_KEYWORDS.csv" AS csvLine
459  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
460  CREATE(movie)-[:HAS_KEYWORDS]->(mi2_info);
461
462  USING PERIODIC COMMIT 500
463  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_LANGUAGE.csv" AS csvLine
464  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
465
466  USING PERIODIC COMMIT 5000
467  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_LANGUAGE.csv" AS csvLine
468  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
469  CREATE(movie)-[:EXIST_IN_LANG]->(mi2_info);
470
471  USING PERIODIC COMMIT 500
472  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_LOCATION.csv" AS csvLine
473  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
474
475  USING PERIODIC COMMIT 5000
476  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_LOCATION.csv" AS csvLine
477  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
478  CREATE(movie)-[:HAS_LOCATIONS]->(mi2_info);
479
480  USING PERIODIC COMMIT 500
481  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_MISC.csv" AS csvLine
482  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
483
484  USING PERIODIC COMMIT 5000
485  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_MISC.csv" AS csvLine
486  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
487  CREATE(movie)-[:WITH_MISC_COMPANIES]->(mi2_info);
488
489  USING PERIODIC COMMIT 500
490  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_PRODUCTION_COMPANIES.csv" AS
     csvLine
491  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
492
493  USING PERIODIC COMMIT 5000
494  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_PRODUCTION_COMPANIES.csv" AS
     csvLine
495  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
496  CREATE(movie)-[:WITH_PRODUCTION_COMPANIES]->(mi2_info);
497
498  USING PERIODIC COMMIT 500
499  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_RELEASE_DATES.csv" AS csvLine
500  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
501
502  USING PERIODIC COMMIT 5000
503  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_RELEASE_DATES.csv" AS csvLine
504  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
505  CREATE(movie)-[:RELEASE_DATE]->(mi2_info);
506
507  USING PERIODIC COMMIT 500
508  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_RUNTIMES.csv" AS csvLine
509  MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
510
511  USING PERIODIC COMMIT 5000
512  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_RUNTIMES.csv" AS csvLine
513  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
     toInt(csvLine.MI2_INFOTYPE_ID)})
514  CREATE(movie)-[:RUNTIME]->(mi2_info);
515
516  USING PERIODIC COMMIT 500
517  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_SOUND_MIX.csv" AS csvLine
```

Abbildung 6.13: Import mit Cypher

```
518 MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
519
520 USING PERIODIC COMMIT 5000
521 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_SOUND_MIX.csv" AS csvLine
522 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
    toInt(csvLine.MI2_INFOTYPE_ID)})
523 CREATE(movie)-[:SOUND_MIX]->(mi2_info);
524
525 USING PERIODIC COMMIT 500
526 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_SPECIAL_EFFECTS.csv" AS csvLine
527 MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
528
529 USING PERIODIC COMMIT 5000
530 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_SPECIAL_EFFECTS.csv" AS csvLine
531 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
    toInt(csvLine.MI2_INFOTYPE_ID)})
532 CREATE(movie)-[:SPECIAL_EFFECTS_MADE_BY]->(mi2_info);
533
534 USING PERIODIC COMMIT 500
535 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_TECH_INFO.csv" AS csvLine
536 MERGE(p:MI2_INFO {mi2_info_id: toInt(csvLine.MI2_INFOTYPE_ID), info: csvLine.INFO});
537
538 USING PERIODIC COMMIT 5000
539 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_MI2_TECH_INFO.csv" AS csvLine
540 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(mi2_info:MI2_INFO { mi2_info_id:
    toInt(csvLine.MI2_INFOTYPE_ID)})
541 CREATE(movie)-[:TECH_INFO]->(mi2_info);
542
543
544
545
546
547
548 IMPORT KNOTEN PLOT MIT RELATIONS
549
550 USING PERIODIC COMMIT 500
551 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_PLOT.csv" AS csvLine
552 MERGE(p:PLOT {plotid: toInt(csvLine.PLOT_ID), info: csvLine.PLOT_TEXT, author:
    csvLine.NOTE});
553
554 USING PERIODIC COMMIT 5000
555 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_PLOT.csv" AS csvLine
556 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(plot:PLOT { plotid:
    toInt(csvLine.PLOT_ID)})
557 CREATE(movie)-[:HAS_PLOT]->(plot);
558
559
560
561
562
563 IMPORT KNOTEN AKA_TITLE MIT RELATIONS
564
565 USING PERIODIC COMMIT 500
566 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_AKA_TITLE.csv" AS csvLine
567 MERGE(p:AKA_TITLE {titleid: toInt(csvLine.TITLE_ID), title: csvLine.MOVIE_AKA_TITLE});
568
569 USING PERIODIC COMMIT 5000
570 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_AKA_TITLE.csv" AS csvLine
571 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(aka_title:AKA_TITLE { titleid:
    toInt(csvLine.TITLE_ID)})
572 CREATE(movie)-[:HAS_TITLE {language: csvLine.LANGUAGE}]->(aka_title);
573
574
575
576
577
578 RELATIONEN ERSTELLEN ZWISCHEN MOVIE UND MOVIE
579
580 USING PERIODIC COMMIT 5000
581 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_ALTERNATE_LANG.csv" AS csvLine
582 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
583 CREATE(movie)-[:ALT_LANG]->(moviedest)
```

Abbildung 6.14: Import mit Cypher

```
584
585 USING PERIODIC COMMIT 5000
586 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_EDITED_FROM.csv" AS csvLine
587 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
588 CREATE(movie)-[:EDITED_FROM]->(moviedest)
589
590 USING PERIODIC COMMIT 5000
591 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_EDITED_INTO.csv" AS csvLine
592 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
593 CREATE(movie)-[:EDITED_INTO]->(moviedest)
594
595 USING PERIODIC COMMIT 5000
596 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_FEATURED_IND.csv" AS csvLine
597 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
598 CREATE(movie)-[:FEATURED_IN]->(moviedest)
599
600 USING PERIODIC COMMIT 5000
601 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_FEATURES.csv" AS csvLine
602 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
603 CREATE(movie)-[:FEATURES]->(moviedest)
604
605 USING PERIODIC COMMIT 5000
606 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_FOLLOWED_BY.csv" AS csvLine
607 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
608 CREATE(movie)-[:FOLLOWED_BY]->(moviedest)
609
610 USING PERIODIC COMMIT 5000
611 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_FOLLOWS.csv" AS csvLine
612 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
613 CREATE(movie)-[:FOLLOWS]->(moviedest)
614
615 USING PERIODIC COMMIT 5000
616 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_REFERENCED_IN.csv" AS csvLine
617 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
618 CREATE(movie)-[:REFERENCED_IN]->(moviedest)
619
620 USING PERIODIC COMMIT 5000
621 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_REFERENCES.csv" AS csvLine
622 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
623 CREATE(movie)-[:REFERENCES]->(moviedest)
624
625 USING PERIODIC COMMIT 5000
626 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_REMADE_AS.csv" AS csvLine
627 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
628 CREATE(movie)-[:REMADE_AS]->(moviedest)
629
630 USING PERIODIC COMMIT 5000
631 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_REMAKE_OF.csv" AS csvLine
632 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
633 CREATE(movie)-[:REMAKE_OF]->(moviedest)
634
635 USING PERIODIC COMMIT 5000
636 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_SPIN_OFF.csv" AS csvLine
637 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
638 CREATE(movie)-[:SPIN_OFF]->(moviedest)
639
640 USING PERIODIC COMMIT 5000
641 LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_SPIN_OFF_FROM.csv" AS csvLine
642 MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
    toInt(csvLine.DEST_MOVIEID)})
643 CREATE(movie)-[:SPIN_OFF_FROM]->(moviedest)
644
```

Abbildung 6.15: Import mit Cypher

```
645  USING PERIODIC COMMIT 5000
646  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_SPOOFED_IN.csv" AS csvLine
647  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
     toInt(csvLine.DEST_MOVIEID)})
648  CREATE(movie)-[:SPOOFED_IN]->(moviedest)
649
650  USING PERIODIC COMMIT 5000
651  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_SPOOFS.csv" AS csvLine
652  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
     toInt(csvLine.DEST_MOVIEID)})
653  CREATE(movie)-[:SPOOFS]->(moviedest)
654
655  USING PERIODIC COMMIT 5000
656  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_LINK_VERSION_OF.csv" AS csvLine
657  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(moviedest:MOVIE { id:
     toInt(csvLine.DEST_MOVIEID)})
658  CREATE(movie)-[:VERSION_OF]->(moviedest)
659
660
661
662
663
664
665
666
667  ERSTELLEN RELATIONS VON SERIEN UND EPISODEN
668
669  USING PERIODIC COMMIT 5000
670  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_EPISODE.csv" AS csvLine
671  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)}),(episodeof:MOVIE { id:
     toInt(csvLine.EPISODEOF)})
672  FOREACH(ignoreMe IN CASE WHEN csvLine.SEASON <> "" OR csvLine.EPISODE <> "" THEN [1]
     ELSE [0] END |
673  CREATE(movie)-[:EPISODEOF { episode: csvLine.EPISODE, season: csvLine.SEASON,
     episodeyear: csvLine.EPISODESYEAR}]->(episodeof)-[:HAS_EPISODE { episode:
     csvLine.EPISODE, season: csvLine.SEASON, episodeyear: csvLine.EPISODESYEAR}]->(movie))
674
675
676
677
678  SETZEN VON NEUEN PROPERTIES BEI MOVIES
679
680  USING PERIODIC COMMIT 5000
681  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_TAGLINES.csv" AS csvLine
682  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)})
683  SET movie.tagline = csvLine.TAGLINE;
684
685
686
687
688
689  USING PERIODIC COMMIT 5000
690  LOAD CSV WITH HEADERS FROM "file:///DBNew/MOVIE_RATING.csv" AS csvLine
691  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)})
692  SET movie.rating = csvLine.RATING, movie.votes = csvLine.VOTES,
     movie.votes_distribution = csvLine.VOTES_DISTRIBUTION;
693
694
695
696
697  USING PERIODIC COMMIT 5000
698  LOAD CSV WITH HEADERS FROM "file:///DBNew/MPAA_RATING.csv" AS csvLine
699  MATCH(movie:MOVIE { id: toInt(csvLine.MOVIE_ID)})
700  SET movie.mpaa_rating = csvLine.RATED_TEXT;
```

Abbildung 6.16: Import mit Cypher

```
1   KNOTEN BIO ERSTELLEN
2
3   SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
4   FROM IMDB_BIO b
5   INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
6   ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
7   WHERE c.BIO_INFOTYPE_TEXT = 'agent address';
8
9   SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
10  FROM IMDB_BIO b
11  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
12  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
13  WHERE c.BIO_INFOTYPE_TEXT = 'articles';
14
15  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
16  FROM IMDB_BIO b
17  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
18  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
19  WHERE c.BIO_INFOTYPE_TEXT = 'biographical movies';
20
21  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
22  FROM IMDB_BIO b
23  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
24  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
25  WHERE c.BIO_INFOTYPE_TEXT = 'birth date';
26
27  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
28  FROM IMDB_BIO b
29  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
30  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
31  WHERE c.BIO_INFOTYPE_TEXT = 'birth name';
32
33  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
34  FROM IMDB_BIO b
35  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
36  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
37  WHERE c.BIO_INFOTYPE_TEXT = 'birth notes';
38
39  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
40  FROM IMDB_BIO b
41  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
42  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
43  WHERE c.BIO_INFOTYPE_TEXT = 'books';
44
45  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
46  FROM IMDB_BIO b
47  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
48  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
49  WHERE c.BIO_INFOTYPE_TEXT = 'death date';
50
51  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
52  FROM IMDB_BIO b
53  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
54  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
55  WHERE c.BIO_INFOTYPE_TEXT = 'death note';
56
57  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
58  FROM IMDB_BIO b
59  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
60  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
61  WHERE c.BIO_INFOTYPE_TEXT = 'height';
62
63  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
```

Abbildung 6.17: Export mit SQL

```
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
64  FROM IMDB_BIO b
65  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
66  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
67  WHERE c.BIO_INFOTYPE_TEXT = 'interviews';
68
69  SELECT b.TEXT AS BIO_TEXT, b.ID AS BIO_ID, b.PERSON_ID AS PERSON_ID,
    c.BIO_INFOTYPE_TEXT AS INFOTYPE
70  FROM IMDB_BIO b
71  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BIO c
72  ON b.BIO_INFOTYPE_ID = c.BIO_INFOTYPE_ID
73  WHERE c.BIO_INFOTYPE_TEXT = 'magazine covers';
74
75
76
77  KNOTEN ALIAS ERSTELLEN
78
79  SELECT p.PERSON_AKA_NAME AS PERSON_AKA_NAME, p.ID AS AKA_ID, p.PERSON_ID AS
    PERSON_ID, d.AKA_NAME_TYPE_TEXT AS NAME_TYPE
80  FROM IMDB.IMDB_AKA_NAMES p
81  INNER JOIN IMDB_SLV_AKA_NAME_TYPE d
82  ON p.AKA_NAME_TYPE_ID = d.AKA_NAME_TYPE_ID
83  WHERE AKA_NAME_TYPE_TEXT = 'aka';
84
85  SELECT p.PERSON_AKA_NAME AS PERSON_AKA_NAME, p.ID AS AKA_ID, p.PERSON_ID AS
    PERSON_ID, d.AKA_NAME_TYPE_TEXT AS NAME_TYPE
86  FROM IMDB.IMDB_AKA_NAMES p
87  INNER JOIN IMDB_SLV_AKA_NAME_TYPE d
88  ON p.AKA_NAME_TYPE_ID = d.AKA_NAME_TYPE_ID
89  WHERE AKA_NAME_TYPE_TEXT = 'birth name';
90
91  SELECT p.PERSON_AKA_NAME AS PERSON_AKA_NAME, p.ID AS AKA_ID, p.PERSON_ID AS
    PERSON_ID, d.AKA_NAME_TYPE_TEXT AS NAME_TYPE
92  FROM IMDB.IMDB_AKA_NAMES p
93  INNER JOIN IMDB_SLV_AKA_NAME_TYPE d
94  ON p.AKA_NAME_TYPE_ID = d.AKA_NAME_TYPE_ID
95  WHERE AKA_NAME_TYPE_TEXT = 'nick name';
96
97
98
99  KNOTEN UND RELATION PERSON ERSTELLEN
100
101 SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
    AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
102 FROM (IMDB_CAST p
103 INNER JOIN IMDB_PERSON e
104 ON p.PERSON_ID = e.ID)
105 INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
106 ON p.ROLETYPE_ID = b.ROLETYPE_ID
107 WHERE b.ROLETYPE_TEXT = 'actor';
108
109 SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
    AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
110 FROM (IMDB_CAST p
111 INNER JOIN IMDB_PERSON e
112 ON p.PERSON_ID = e.ID)
113 INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
114 ON p.ROLETYPE_ID = b.ROLETYPE_ID
115 WHERE b.ROLETYPE_TEXT = 'actress';
116
117 SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
    AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
118 FROM (IMDB_CAST p
119 INNER JOIN IMDB_PERSON e
120 ON p.PERSON_ID = e.ID)
121 INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
122 ON p.ROLETYPE_ID = b.ROLETYPE_ID
123 WHERE b.ROLETYPE_TEXT = 'cinematographer';
124
125 SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
    AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
126 FROM (IMDB_CAST p
127 INNER JOIN IMDB_PERSON e
```

Abbildung 6.18: Export mit SQL

```
128  ON p.PERSON_ID = e.ID)
129  INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
130  ON p.ROLETYPE_ID = b.ROLETYPE_ID
131  WHERE b.ROLETYPE_TEXT = 'composer';
132
133  SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
     AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
134  FROM (IMDB_CAST p
135  INNER JOIN IMDB_PERSON e
136  ON p.PERSON_ID = e.ID)
137  INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
138  ON p.ROLETYPE_ID = b.ROLETYPE_ID
139  WHERE b.ROLETYPE_TEXT = 'costume designer';
140
141  SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
     AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
142  FROM (IMDB_CAST p
143  INNER JOIN IMDB_PERSON e
144  ON p.PERSON_ID = e.ID)
145  INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
146  ON p.ROLETYPE_ID = b.ROLETYPE_ID
147  WHERE b.ROLETYPE_TEXT = 'director';
148
149  SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
     AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
150  FROM (IMDB_CAST p
151  INNER JOIN IMDB_PERSON e
152  ON p.PERSON_ID = e.ID)
153  INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
154  ON p.ROLETYPE_ID = b.ROLETYPE_ID
155  WHERE b.ROLETYPE_TEXT = 'editor';
156
157  SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
     AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
158  FROM (IMDB_CAST p
159  INNER JOIN IMDB_PERSON e
160  ON p.PERSON_ID = e.ID)
161  INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
162  ON p.ROLETYPE_ID = b.ROLETYPE_ID
163  WHERE b.ROLETYPE_TEXT = 'miscellaneous crew';
164
165  SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
     AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
166  FROM (IMDB_CAST p
167  INNER JOIN IMDB_PERSON e
168  ON p.PERSON_ID = e.ID)
169  INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
170  ON p.ROLETYPE_ID = b.ROLETYPE_ID
171  WHERE b.ROLETYPE_TEXT = 'producer';
172
173  SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
     AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
174  FROM (IMDB_CAST p
175  INNER JOIN IMDB_PERSON e
176  ON p.PERSON_ID = e.ID)
177  INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
178  ON p.ROLETYPE_ID = b.ROLETYPE_ID
179  WHERE b.ROLETYPE_TEXT = 'production designer';
180
181  SELECT p.PERSON_ID AS PERSON_ID, p.ROLENAME AS ROLENAME, p.MOVIE_ID AS MOVIEID, p.ID
     AS CASTID, b.ROLETYPE_TEXT AS ROLETYPE, e.FULLNAME AS FULLNAME
182  FROM (IMDB_CAST p
183  INNER JOIN IMDB_PERSON e
184  ON p.PERSON_ID = e.ID)
185  INNER JOIN IMDB.IMDB_SLV_ROLETYPE b
186  ON p.ROLETYPE_ID = b.ROLETYPE_ID
187  WHERE b.ROLETYPE_TEXT = 'writer';
188
189
190
191  RELATION MOVIE MIT MOVIE ERSTELLEN
192
193  SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
```

Abbildung 6.19: Export mit SQL

```
    b.MOVIELINK_TYPE_TEXT AS TYPE
194 FROM IMDB.IMDB_MOVIE_LINKS p
195 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
196 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
197 WHERE b.MOVIELINK_TYPE_TEXT = 'alternate language version of';
198
199 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
200 FROM IMDB.IMDB_MOVIE_LINKS p
201 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
202 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
203 WHERE b.MOVIELINK_TYPE_TEXT = 'edited from';
204
205 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
206 FROM IMDB.IMDB_MOVIE_LINKS p
207 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
208 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
209 WHERE b.MOVIELINK_TYPE_TEXT = 'edited into';
210
211 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
212 FROM IMDB.IMDB_MOVIE_LINKS p
213 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
214 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
215 WHERE b.MOVIELINK_TYPE_TEXT = 'featured in';
216
217 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
218 FROM IMDB.IMDB_MOVIE_LINKS p
219 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
220 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
221 WHERE b.MOVIELINK_TYPE_TEXT = 'features';
222
223 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
224 FROM IMDB.IMDB_MOVIE_LINKS p
225 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
226 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
227 WHERE b.MOVIELINK_TYPE_TEXT = 'followed by';
228
229 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
230 FROM IMDB.IMDB_MOVIE_LINKS p
231 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
232 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
233 WHERE b.MOVIELINK_TYPE_TEXT = 'follows';
234
235 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
236 FROM IMDB.IMDB_MOVIE_LINKS p
237 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
238 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
239 WHERE b.MOVIELINK_TYPE_TEXT = 'referenced in';
240
241 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
242 FROM IMDB.IMDB_MOVIE_LINKS p
243 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
244 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
245 WHERE b.MOVIELINK_TYPE_TEXT = 'references';
246
247 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
248 FROM IMDB.IMDB_MOVIE_LINKS p
249 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
250 ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
251 WHERE b.MOVIELINK_TYPE_TEXT = 'remade as';
252
253 SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
    b.MOVIELINK_TYPE_TEXT AS TYPE
254 FROM IMDB.IMDB_MOVIE_LINKS p
255 INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
```

Abbildung 6.20: Export mit SQL

```
256  ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
257  WHERE b.MOVIELINK_TYPE_TEXT = 'remake of';
258
259  SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
     b.MOVIELINK_TYPE_TEXT AS TYPE
260  FROM IMDB.IMDB_MOVIE_LINKS p
261  INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
262  ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
263  WHERE b.MOVIELINK_TYPE_TEXT = 'spin off';
264
265  SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
     b.MOVIELINK_TYPE_TEXT AS TYPE
266  FROM IMDB.IMDB_MOVIE_LINKS p
267  INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
268  ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
269  WHERE b.MOVIELINK_TYPE_TEXT = 'spin off from';
270
271  SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
     b.MOVIELINK_TYPE_TEXT AS TYPE
272  FROM IMDB.IMDB_MOVIE_LINKS p
273  INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
274  ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
275  WHERE b.MOVIELINK_TYPE_TEXT = 'spoofed in';
276
277  SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
     b.MOVIELINK_TYPE_TEXT AS TYPE
278  FROM IMDB.IMDB_MOVIE_LINKS p
279  INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
280  ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
281  WHERE b.MOVIELINK_TYPE_TEXT = 'spoofs';
282
283  SELECT p.SRC_MOVIE_ID AS MOVIE_ID, p.DEST_MOVIE_ID AS DEST_MOVIEID,
     b.MOVIELINK_TYPE_TEXT AS TYPE
284  FROM IMDB.IMDB_MOVIE_LINKS p
285  INNER JOIN IMDB.IMDB_SLV_MOVIELINK_TYPE b
286  ON p.MOVIELINK_TYPE_ID = b.MOVIELINK_TYPE_ID
287  WHERE b.MOVIELINK_TYPE_TEXT = 'version of';
288
289
290
291  KNOTEN MI2_INFO
292
293  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
294  FROM IMDB.IMDB_MOVIE_INFO_2 p
295  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
296  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
297  WHERE b.MI2_INFOTYPE_TEXT = 'certificates';
298
299  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
300  FROM IMDB.IMDB_MOVIE_INFO_2 p
301  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
302  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
303  WHERE b.MI2_INFOTYPE_TEXT = 'color info';
304
305  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
306  FROM IMDB.IMDB_MOVIE_INFO_2 p
307  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
308  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
309  WHERE b.MI2_INFOTYPE_TEXT = 'countries';
310
311  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
312  FROM IMDB.IMDB_MOVIE_INFO_2 p
313  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
314  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
315  WHERE b.MI2_INFOTYPE_TEXT = 'distributors';
316
317  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
318  FROM IMDB.IMDB_MOVIE_INFO_2 p
```

Abbildung 6.21: Export mit SQL

```
319  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
320  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
321  WHERE b.MI2_INFOTYPE_TEXT = 'genres';
322
323  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
324  FROM IMDB.IMDB_MOVIE_INFO_2 p
325  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
326  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
327  WHERE b.MI2_INFOTYPE_TEXT = 'keywords';
328
329  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
330  FROM IMDB.IMDB_MOVIE_INFO_2 p
331  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
332  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
333  WHERE b.MI2_INFOTYPE_TEXT = 'language';
334
335  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
336  FROM IMDB.IMDB_MOVIE_INFO_2 p
337  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
338  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
339  WHERE b.MI2_INFOTYPE_TEXT = 'locations';
340
341  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
342  FROM IMDB.IMDB_MOVIE_INFO_2 p
343  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
344  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
345  WHERE b.MI2_INFOTYPE_TEXT = 'miscellaneous companies';
346
347  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
348  FROM IMDB.IMDB_MOVIE_INFO_2 p
349  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
350  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
351  WHERE b.MI2_INFOTYPE_TEXT = 'production companies';
352
353  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
354  FROM IMDB.IMDB_MOVIE_INFO_2 p
355  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
356  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
357  WHERE b.MI2_INFOTYPE_TEXT = 'release dates';
358
359  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
360  FROM IMDB.IMDB_MOVIE_INFO_2 p
361  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
362  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
363  WHERE b.MI2_INFOTYPE_TEXT = 'runtimes';
364
365  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
366  FROM IMDB.IMDB_MOVIE_INFO_2 p
367  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
368  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
369  WHERE b.MI2_INFOTYPE_TEXT = 'sound mix';
370
371  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
372  FROM IMDB.IMDB_MOVIE_INFO_2 p
373  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
374  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
375  WHERE b.MI2_INFOTYPE_TEXT = 'special effects companies';
376
377  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI2_INFOTYPE_ID,
     b.MI2_INFOTYPE_TEXT AS INFO_MI2
378  FROM IMDB.IMDB_MOVIE_INFO_2 p
379  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI2 b
380  ON p.MI2_INFOTYPE_ID = b.MI2_INFOTYPE_ID
381  WHERE b.MI2_INFOTYPE_TEXT = 'tech info';
```

Abbildung 6.22: Export mit SQL

```
382
383
384
385  KNOTEN PLOT
386
387  SELECT p.MOVIE_ID AS MOVIE_ID, p.ID AS PLOT_ID, p.TEXT AS PLOT, p.ID AS PLOT_TEXT,
     p.NOTE AS NOTE
388  FROM IMDB_PLOT p;
389
390
391  KNOTEN MI_INFO
392
393  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI_INFOTYPE,
     m.MI_INFOTYPE_TEXT AS INFO_MI
394  FROM IMDB_MOVIE_INFO p
395  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI m
396  ON p.MI_INFOTYPE_ID = m.MI_INFOTYPE_ID
397  WHERE m.MI_INFOTYPE_TEXT = 'alternate version' AND p.ID IS NOT NULL AND
     m.MI_INFOTYPE_TEXT IS NOT NULL AND p.MOVIE_ID IS NOT NULL;
398
399  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI_INFOTYPE,
     m.MI_INFOTYPE_TEXT AS INFO_MI
400  FROM IMDB_MOVIE_INFO p
401  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI m
402  ON p.MI_INFOTYPE_ID = m.MI_INFOTYPE_ID
403  WHERE m.MI_INFOTYPE_TEXT = 'crazy credits' AND p.ID IS NOT NULL AND
     m.MI_INFOTYPE_TEXT IS NOT NULL AND p.MOVIE_ID IS NOT NULL;
404
405  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI_INFOTYPE,
     m.MI_INFOTYPE_TEXT AS INFO_MI
406  FROM IMDB_MOVIE_INFO p
407  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI m
408  ON p.MI_INFOTYPE_ID = m.MI_INFOTYPE_ID
409  WHERE m.MI_INFOTYPE_TEXT = 'goofs' AND p.ID IS NOT NULL AND m.MI_INFOTYPE_TEXT IS
     NOT NULL AND p.MOVIE_ID IS NOT NULL;
410
411  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI_INFOTYPE,
     m.MI_INFOTYPE_TEXT AS INFO_MI
412  FROM IMDB_MOVIE_INFO p
413  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI m
414  ON p.MI_INFOTYPE_ID = m.MI_INFOTYPE_ID
415  WHERE m.MI_INFOTYPE_TEXT = 'quotes' AND p.ID IS NOT NULL AND m.MI_INFOTYPE_TEXT IS
     NOT NULL AND p.MOVIE_ID IS NOT NULL;
416
417  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI_INFOTYPE,
     m.MI_INFOTYPE_TEXT AS INFO_MI
418  FROM IMDB_MOVIE_INFO p
419  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI m
420  ON p.MI_INFOTYPE_ID = m.MI_INFOTYPE_ID
421  WHERE m.MI_INFOTYPE_TEXT = 'soundtrack' AND p.ID IS NOT NULL AND m.MI_INFOTYPE_TEXT
     IS NOT NULL AND p.MOVIE_ID IS NOT NULL;
422
423  SELECT p.MOVIE_ID AS MOVIE_ID, p.INFO AS INFO, p.ID AS MI_INFOTYPE,
     m.MI_INFOTYPE_TEXT AS INFO_MI
424  FROM IMDB_MOVIE_INFO p
425  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_MI m
426  ON p.MI_INFOTYPE_ID = m.MI_INFOTYPE_ID
427  WHERE m.MI_INFOTYPE_TEXT = 'trivia' AND p.ID IS NOT NULL AND m.MI_INFOTYPE_TEXT IS
     NOT NULL AND p.MOVIE_ID IS NOT NULL;
428
429
430
431
432  KNOTEN BUSINESS
433
434  SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
     m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
435  FROM IMDB_BUSINESS p
436  INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
437  ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
438  WHERE m.BUSINESS_INFOTYPE_TEXT = 'admissions';
439
440  SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
```

Abbildung 6.23: Export mit SQL

```
    m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
441 FROM IMDB_BUSINESS p
442 INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
443 ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
444 WHERE m.BUSINESS_INFOTYPE_TEXT = 'budget';
445
446 SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
    m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
447 FROM IMDB_BUSINESS p
448 INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
449 ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
450 WHERE m.BUSINESS_INFOTYPE_TEXT = 'copyright holder';
451
452 SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
    m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
453 FROM IMDB_BUSINESS p
454 INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
455 ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
456 WHERE m.BUSINESS_INFOTYPE_TEXT = 'filming dates';
457
458 SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
    m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
459 FROM IMDB_BUSINESS p
460 INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
461 ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
462 WHERE m.BUSINESS_INFOTYPE_TEXT = 'gross';
463
464 SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
    m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
465 FROM IMDB_BUSINESS p
466 INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
467 ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
468 WHERE m.BUSINESS_INFOTYPE_TEXT = 'opening weekend';
469
470 SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
    m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
471 FROM IMDB_BUSINESS p
472 INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
473 ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
474 WHERE m.BUSINESS_INFOTYPE_TEXT = 'production dates';
475
476 SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
    m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
477 FROM IMDB_BUSINESS p
478 INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
479 ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
480 WHERE m.BUSINESS_INFOTYPE_TEXT = 'rentals';
481
482 SELECT p.MOVIE_ID AS MOVIE_ID, p.TEXT AS BUSINESS_TEXT, p.ID AS BUSINESS_ID,
    m.BUSINESS_INFOTYPE_TEXT AS BUSINESS_INFO
483 FROM IMDB_BUSINESS p
484 INNER JOIN IMDB.IMDB_SLV_INFOTYPE_BUSINESS m
485 ON p.BUSINESS_INFOTYPE_ID = m.BUSINESS_INFOTYPE_ID
486 WHERE m.BUSINESS_INFOTYPE_TEXT = 'weekend gross';
487
488
489 KNOTEN AKA_TITLE
490
491 SELECT NVL(RTRIM(LTRIM(regexp_substr(j.NOTE, '([^()]+)'), ') '),')'), j.NOTE) AS
    LANGUAGE, j.MOVIE_AKA_TITLE AS MOVIE_AKA_TITLE, j.MOVIE_ID AS MOVIE_ID, j.ID AS
    TITLE_ID
492 FROM IMDB.IMDB_AKA_TITLES j;
493
494
495 SET RATINGS VORBEREITUNG
496
497 SELECT p.MOVIE_ID AS MOVIE_ID, p.VOTES_DISTRIBUTION AS VOTES_DISTRIBUTION,p.VOTES AS
    VOTES, p.RATING AS RATING, p.ID AS RATING_ID
498 FROM IMDB.IMDB_RATINGS p;
499
500
501 SET TAGLINES VORBEREITUNG
502
```

Abbildung 6.24: Export mit SQL

```
503  SELECT p.MOVIE_ID AS MOVIE_ID, p.ID AS TAGLINE_ID, p.TAGLINE AS TAGLINE
504  FROM IMDB_TAGLINES p;
505
506
507  KNOTEN MOVIE ERSTELLEN
508
509  SELECT p.ID AS MOVIE_ID, p.FULLTITLE AS FULLTITLE, p.TITLE AS TITLE, p.YEAR AS YEAR,
     m.MOVIE_KIND_TEXT AS KIND
510  FROM IMDB_MOVIE p
511  INNER JOIN IMDB.IMDB_SLV_KIND_MOVIE m
512  ON p.MOVIE_KIND_ID = m.MOVIE_KIND_ID;
513
514
515  RELATION MOVIE UND EPISODE EXPORT
516
517  SELECT p.MOVIE_KIND_TEXT AS KIND, m.EPISODEOF_MOVIE_ID AS EPISODEOF, m.ID AS
     MOVIE_ID, m.SEASON AS SEASON, m.EPISODE AS EPISODE, m.EPISODESYEAR AS EPISODESYEAR
518  FROM IMDB_MOVIE m
519  INNER JOIN IMDB.IMDB_SLV_KIND_MOVIE p
520  ON m.MOVIE_KIND_ID = p.MOVIE_KIND_ID
521  WHERE p.MOVIE_KIND_TEXT = 'episode';
522
523
524  MPAA_RATINGS VORBEREITUNG
525
526  SELECT m.ID AS MOVIE_ID, m.FULLTITLE AS FULLTITLE, r.TEXT AS PARENTAL_CONTROL
527  FROM IMDB.IMDB_MOVIE m, IMDB.IMDB_MPAA_RATING_REASONS r
528  WHERE r.MOVIE_ID = m.ID;
```

Abbildung 6.25: Export mit SQL

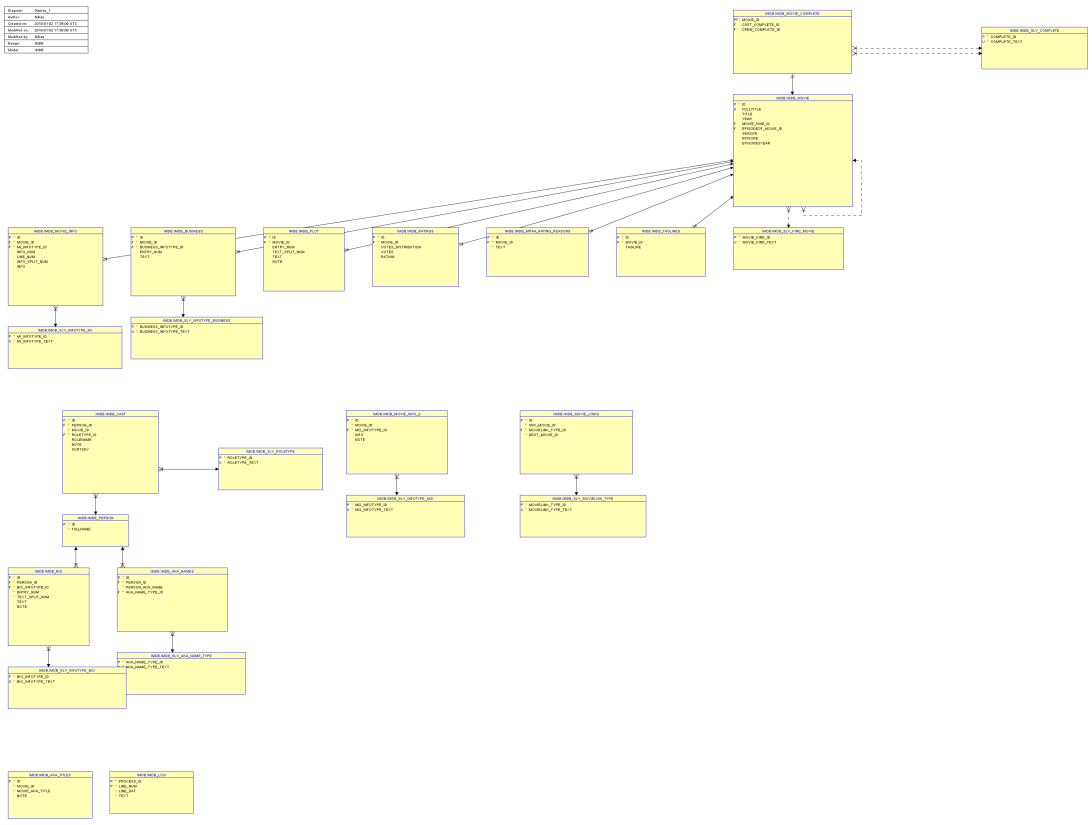

Abbildung 6.26: Relationale Modell IMDB